TAO Developer’s Guide:
Volume 1 of 2

OCI TAO Version 1.4a
OCI Part Number 541-01 of the two-volume set (OCI Part Number 540-01)
Your feedback on, or your submission of content to, this documentation is appreciated. Please contact us at either office location via phone or fax, or e-mail your feedback to: techpubs@ociweb.com.

**How to contact us:**

Object Computing, Inc. (Corporate Office)  
12140 Woodcrest Executive Drive, Suite 250  
St. Louis, MO 63141  
+1.314.579.0066 Voice  
+1.314.579.0065 Fax

Object Computing, Inc.  
64 E Broadway Road, Suite 160  
Tempe, AZ 85282  
+1.480.752.0042 Voice  
+1.480.752.0076 Fax

Support: support@ociweb.com  
Training: training@ociweb.com  
Sales: sales@ociweb.com  
Internet: www.ociweb.com
Contents

Volume 1

Contents ................................................................. vii
Foreword ............................................................... xix
Preface ................................................................. xxv
Detailed Licensing Terms .......................................... xxxix

Part 1 Introduction to TAO Programming ........................ 1

Chapter 1 Introduction ................................................ 3
Design Goals ......................................................... 4
Development History ............................................... 6
Architecture of TAO ................................................... 8
CORBA Compliance ................................................ 12
## Contents

<table>
<thead>
<tr>
<th>Chapter 2 Building ACE and TAO</th>
<th>Chapter 3 Getting Started</th>
<th>Chapter 4 The Makefile, Project, and Workspace Creator (MPC)</th>
<th>Part 2 Features of TAO</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Performance and Real-Time Support</td>
<td>Introduction</td>
<td>Introduction</td>
<td>TAO IDL Compiler</td>
</tr>
<tr>
<td>Relationship Between ACE and TAO</td>
<td>Where to Get ACE and TAO</td>
<td>Setting Up Your Environment</td>
<td>Executables</td>
</tr>
<tr>
<td></td>
<td>System Requirements</td>
<td>A Simple Example</td>
<td>Output Files Generated</td>
</tr>
<tr>
<td></td>
<td>Make Project Creator (MPC)</td>
<td>Summary</td>
<td>Using TAO IDL Compiler Options</td>
</tr>
<tr>
<td></td>
<td>Configuring Builds</td>
<td></td>
<td>Preprocessing Options</td>
</tr>
</tbody>
</table>

### Chapter 2 Building ACE and TAO
- Introduction ............................................. 17
- Where to Get ACE and TAO .......................... 17
- System Requirements ................................ 18
- Make Project Creator (MPC) ......................... 19
- Configuring Builds .................................. 19

### Chapter 3 Getting Started
- Introduction ............................................. 21
- Setting Up Your Environment ....................... 22
- A Simple Example ...................................... 22
- Summary .................................................. 31

### Chapter 4 The Makefile, Project, and Workspace Creator (MPC)
- Introduction ............................................. 33
- Using MPC ............................................... 34
- Writing MPC and MWC Files ......................... 42
- Adding a New Type ..................................... 65

### Part 2 Features of TAO

### Chapter 5 TAO IDL Compiler
- Introduction ............................................. 83
- Executables .............................................. 84
- Output Files Generated .............................. 84
- Using TAO IDL Compiler Options ................... 86
- Preprocessing Options ............................... 86
- Output File Options .................................. 89
- Starter Implementation Files ....................... 90
- Additional Code Generation Options ............... 92
- The Alternate C++ Mapping for Exceptions ....... 93
## Contents

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Error Handling</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Introduction</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>CORBA System Exceptions</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>CORBA User Exceptions</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Error Handling Without Exception Handling</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>The ACE Exception Handling Macros</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>Synopsis of TAO’s Error Handling Guidelines</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>TAO Minor Codes</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>137</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>CORBA Messaging</th>
<th>139</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Introduction</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>AMI Callback Model</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>Quality of Service Policies</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>Bi-Directional GIOP</td>
<td>180</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 8</th>
<th>Asynchronous Method Handling</th>
<th>185</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Introduction</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>Participants in an AMH Servant</td>
<td>188</td>
</tr>
<tr>
<td></td>
<td>Generating AMH Related Code</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>An AMH Example Program</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>AMH and Advanced CORBA Features</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td>Combining AMH with AMI</td>
<td>202</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 9</th>
<th>Real-Time CORBA</th>
<th>211</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Introduction</td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>Real-Time CORBA Overview</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>Real-Time CORBA Architecture</td>
<td>215</td>
</tr>
</tbody>
</table>
## Contents

Dynamic Scheduling .................................................. 234
TAO’s Implementation of Real-Time CORBA ....................... 246
Client-Propagated Priority Model .................................. 266
Server-Declared Priority Model ..................................... 269
Using the RTScheduling::Current .................................. 273
Real-Time CORBA Examples ......................................... 278

### Chapter 10  **Portable Interceptors**  ........................................ 281
Introduction .................................................................. 281
Using TAO Request Interceptors .................................... 282
Marshaling and the Service Context ................................. 299
IOR Interceptors .......................................................... 305
The PortableInterceptor::Current ................................... 311
Summary ..................................................................... 316

### Chapter 11  **Value Types** .................................................. 317
Introduction .................................................................. 317
Uses for Value Types .................................................... 318
Defining Value Types in IDL ........................................... 319
A Value Type Example .................................................. 320
An Example using Value Types as Events ......................... 325
Value Types and Inheritance .......................................... 328
TAO Compliance .......................................................... 330

### Chapter 12  **Smart Proxies** .................................................. 331
Introduction .................................................................. 331
Smart Proxy Use Cases .................................................. 333
TAO’s Smart Proxy Framework ....................................... 334
Writing and Using Smart Proxy Classes ......................... 340
Linking Your Application ............................................... 343
A Smart Proxy Example ................................................. 344

### Chapter 13  **Local Interfaces** .................................................. 353
Introduction .................................................................. 353
Contents

C++ Mapping for LocalObject ............................................. 354
Changing Existing Interfaces to Local Interfaces ......................... 355
Example: ServantLocator ................................................. 356

Chapter 14 IOR Table ....................................................... 365
   Introduction ......................................................... 365
   IOR Table .......................................................... 366
   Locator ............................................................ 368

Chapter 15 Using Pluggable Protocols .................................. 371
   Introduction ......................................................... 371
   Protocol Introduction .............................................. 372
   Protocols Provided with TAO ....................................... 374
   Building the Protocol Libraries ................................... 375
   Loading Pluggable Protocols ...................................... 375
   IIOP ............................................................... 376
   IIOP_Lite .......................................................... 379
   UIOP .............................................................. 380
   SHMIOP ............................................................ 382
   DIOP .............................................................. 385
   SSLIOP ............................................................. 388
   MIOP/UIPMC ..................................................... 389
   HTIOP ............................................................. 393
   SCIOP ............................................................. 396
   Combining Protocols ............................................... 398

Chapter 16 Developing Pluggable Protocols .............................. 401
   Introduction ......................................................... 401
   Pluggable Protocol Requirements ................................ 402
   Overview of Pluggable Protocols in TAO ......................... 403
   Details of the Pluggable-Protocol Framework ....................... 414
   Pluggable Protocol Summary ....................................... 526

Chapter 17 Multithreading with TAO .................................... 527
Contents

Introduction ..................................................... 527
Overview of Client/Server Roles in CORBA .............. 529
Multithreading in the Server ................................. 533
Multithreading in the Client .................................. 571
Summary ........................................................ 588

Part 3  Run-time Configuration of TAO 589

Chapter 18 Configuring TAO Clients and Servers ......... 591
Introduction ..................................................... 591
Patterns and Components for Configuring TAO Clients and Servers .............. 592
The ACE Service Configurator ............................. 594
Service Configurator Control Options ...................... 597
The ACE Service Configurator Framework ................. 598
XML Service Configurator ................................... 603
Service Objects ............................................... 606
ACE Service Manager ......................................... 615
Summary ........................................................ 616

Chapter 19 ORB Initialization Options ................... 617
Introduction ..................................................... 617
Interface Definition ........................................... 618
Controlling Service Configurator Behavior ................. 621
Controlling Debugging Information ......................... 622
Optimizing Request Processing ............................ 622
Connection Management and Protocol Selection .......... 623
Miscellaneous Options ....................................... 625
Option Descriptions .......................................... 625

Chapter 20 Resource Factory ............................... 689
Introduction ..................................................... 689
Interface Definition ........................................... 690
Resource Factory for Qt GUI Toolkit ....................... 707
Resource Factory for X Windowing Toolkit ................. 709
Contents

Advanced Resource Factory ........................................... 710
Resource Factory Options ............................................ 713
Advanced Resource Factory Options ................................. 730

Chapter 21  Server Strategy Factory .................................... 739
Introduction ................................................................... 739
Interface Definition ...................................................... 740
Default Server Strategy Factory Options ......................... 748

Chapter 22  Client Strategy Factory .................................... 763
Introduction ................................................................... 763
Interface Definition ...................................................... 764
Client Strategy Factory Options ...................................... 769

Volume 2

Part 4  TAO Services ......................................................... 779

Chapter 23  TAO Services Overview ................................. 781
Introduction ................................................................... 781
Customizing Access to the Services ................................. 782
TAO’s ORB Services Libraries ........................................ 783
Locating Service Objects .............................................. 786

Chapter 24  Naming Service ................................................ 791
Introduction ................................................................... 791
Resolving the Naming Service ........................................ 793
Naming Service Example ............................................... 794
Object URLs .................................................................. 801
The NamingContextExt Interface ..................................... 807
TAO-Specific Naming Service Classes ......................... 810
Naming Service Utilities .............................................. 814
Contents

Naming Service Command Line Options ............................................. 818

Chapter 25  **Event Service** ............................................................... 825
  Introduction ................................................................. 825
  Overview of the Event Service .............................................. 826
  TAO’s Event Channel Implementation ..................................... 827
  How to Use the Event Service .............................................. 827
  CosEvent_Service Command Line Options ............................... 845
  Event Channel Resource Factory ........................................... 846

Chapter 26  **Real-Time Event Service** ............................................. 871
  Introduction ................................................................. 871
  Overview of the TAO Real-Time Event Service ....................... 871
  Using the TAO Real-Time Event Service ................................ 874
  Event_Service Command Line Options ................................... 913
  Event Channel Resource Factory ........................................... 915
  The IIOP Gateway Resource Factory .................................... 941

Chapter 27  **Notification Service** .................................................. 949
  Introduction ................................................................. 949
  Notification Service Architecture ....................................... 950
  Notification Service Features ............................................ 951
  Using the Notification Service ........................................... 975
  Compatibility with the Event Service .................................. 1021
  Notify_Service Command Line Options ................................ 1021
  Notification Service Configuration Options ......................... 1022

Chapter 28  **Interface Repository** ................................................ 1049
  Introduction ................................................................. 1049
  Using the Interface Repository ......................................... 1050
  TAO’s Interface Repository Implementation .......................... 1051
  Example IFR Client ........................................................ 1055

Chapter 29  **TAO Security** ............................................................... 1065
## Contents

Preface ................................................. 1065  
Introduction ....................................... 1067  
Introduction to CORBA Security .................. 1068  
Secure Sockets Layer Protocol .................... 1087  
Working with Certificates ....................... 1099  
Building ACE and TAO Security Libraries ....... 1106  
Security Unaware Application .................... 1110  
Security Policy Controlling Application ......... 1118  
Security Policy Enforcing Application .......... 1124  
SSLIOP Factory Options ............................ 1132  

### Chapter 30 Implementation Repository ................. 1137  
  Introduction ...................................... 1137  
The Operation of the ImR ......................... 1139  
Basic Indirection Example ....................... 1141  
Server Start-up .................................... 1145  
Activation Modes ................................... 1150  
Using the ImR and the IOR Table ................. 1151  
ImR and IOR Table Example ...................... 1153  
Advanced Examples ................................ 1159  
Repository Persistence ............................ 1159  
ImR Utility .......................................... 1160  
ImplRepo_Service ................................... 1167  
ImR_Activator ...................................... 1171  

### Chapter 31 Data Distribution Service ................. 1175  
  Introduction ...................................... 1175  
DCPS Overview ..................................... 1176  
TAO DDS Implementation ........................... 1180  
Using DCPS ........................................ 1183  
Data Handling Optimizations ..................... 1195  
QoS Policies ........................................ 1196  
Pluggable Transport Details ..................... 1201  
Using Built-In Topics ............................. 1203
## Contents

DCPS Service Participant Options ........................................... 1206  
dcps_ts.pl Command Line Options ........................................... 1207  
DCPS Information Repository Options ........................................... 1209

### Part 5 CIAO

1211

#### Chapter 32 CIAO and CCM

1213

Introduction ................................................................. 1213  
Example - The Messenger Application ........................................... 1219  
Building CIAO .............................................................. 1345  
DAnCE Executable Reference ............................................... 1353  
CIDL Compiler Reference .................................................. 1356  
IDL3-to-IDL2 Compiler Reference ........................................... 1364  
Future Topics .............................................................. 1370

### Part 6 Appendices

1371

#### Appendix A Configuring ACE/TAO Builds

1373

MPC Features .............................................................. 1373  
GNU Make Build Flags .................................................... 1376  
Using the Build Flags ..................................................... 1379

#### Appendix B Choosing How To Build ACE and TAO

1389

#### Appendix C Building ACE and TAO on UNIX

1391

Building ACE and TAO on a UNIX System ........................................... 1391  
Customizing ACE and TAO Builds ........................................... 1397

#### Appendix D Building ACE and TAO Using Visual C++

1401

Why Build a Custom Version? ............................................... 1401  
Building ACE and TAO ..................................................... 1402  
Build Notes .............................................................. 1406

#### Appendix E Building ACE and TAO with Borland C++

1409
# Contents

Building ACE and TAO with Borland C++ .......................... 1409

## Appendix F Using ACE and TAO with VxWorks ......................... 1415
Kernel and System Configuration .................................. 1415
Environment Setup .............................................. 1417

## Appendix G Using ACE and TAO with LynxOS .......................... 1423
Native Compilation ............................................. 1423
Cross-Compilation ............................................ 1425

## Appendix H Testing ACE and TAO on VxWorks and LynxOS .............. 1429
Building the Tests ............................................ 1429
Running the Tests ........................................... 1430

## Appendix I CORBA Compliance ..................................... 1435
Introduction .................................................. 1435
CORBA 3.0 .................................................. 1436
Minimum CORBA ............................................. 1439
Real-Time CORBA ............................................ 1441
C++ Language Mapping ....................................... 1442
Naming Service ............................................... 1443
Notification Service .......................................... 1444
Security Service ............................................. 1445
Data Distribution Service ..................................... 1445
CORBA Component Model ....................................... 1445
DII COE .................................................... 1446

## Appendix J TAO and ACE Contributors ............................... 1449

References .................................................. 1465

Index ..................................................... 1469
One of the most important technology advances over the past two decades has been the evolution and maturation of reusable service-oriented architecture (SOA) and publish/subscribe (pub/sub) middleware based on open standards, such as the OMG CORBA, the CORBA Component Model, and Data Distribution Service. This middleware resides between applications and the underlying operating systems, network protocol stacks, and hardware. At the heart of SOA middleware is the object request broker (ORB), whose primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure to (1) extend the scope of portable software via common industry-wide standards, (2) coordinate how parts of applications are connected and how they interoperate across networked nodes, and (3) ease the integration and interoperability of software artifacts developed by multiple technology suppliers.
When developed and deployed properly, SOA and pub/sub middleware can reduce the cost and risk of developing distributed applications and systems. The right middleware helps to simplify the development of distributed applications by providing a consistent set of capabilities that are closer to application design-level abstractions than to the underlying mechanisms in the computing platforms and communication networks. These higher-level abstractions shield application developers from lower-level, tedious, and error-prone platform details, such as socket-level network programming and multithreading, and help application developers manage system resources, such as memory, network, and CPU resources. Middleware also amortizes software lifecycle costs by (1) leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use and (2) providing a wide range of reusable application-oriented services, such as transactional logging and security, that have proven necessary to operate effectively in distributed environments.

This documentation set from Object Computing, Inc. (OCI) is the fifth installment of an ever growing and improving body of information describing the capabilities and effective usage of

- The **ACE ORB (TAO)**, which is a highly portable, open-source, high-performance, and real-time implementation of the CORBA specification using the C++ frameworks and wrapper facade classes provided by the ACE toolkit.

- The **Component-Integrated ACE ORB (CIAO)**, which extends TAO by providing component-based abstractions using the specification, validation, packaging, configuration, and deployment techniques defined by the Lightweight CORBA Component Model (CCM) and Deployment and Configuration (D&C) specifications.

- **TAO DDS**, which provides a high-performance implementation of the OMG Data Distribution Specification (DDS) specification and defines quality of service (QoS) parameters to configure the middleware and establish contracts that specify how and when information should flow between publishers and subscribers.

In this fifth release from OCI, the TAO, CIAO, and TAO DDS middleware and the agile open-source development process that drives it show their remarkable ability to evolve rapidly, and to continue to lead the field in implementing the latest OMG CORBA, CCM, and DDS specifications and span of platforms supported. Crucial to the success and longevity of our middleware is its robust design based on the patterns and frameworks in ACE that substantially improve its efficiency, predictability, and scalability.

Much of my group's R&D focus over the past two years since the fourth OCI release has focused on optimizing the time and space overhead of ACE and TAO so they can meet more stringent application QoS requirements in a wider range of domains.
The figure above illustrates where optimizations and recent enhancements have been applied to the following TAO components:

- TAO's ORB Core supports deterministic real-time concurrency and dispatching strategies designed to minimize context switching, synchronization, dynamic memory allocation, and data movement.

- TAO's Portable Object Adapter (POA) implementation uses active demultiplexing and perfect hashing optimizations that associate client requests with target objects in constant time, regardless of the number of objects, operations, or nested POAs. Recent work on subsetting TAO 1.4a’s POA has significantly reduced its memory footprint for common use cases.

- TAO includes a highly optimized CORBA IIOP protocol engine and an IDL compiler that generates compiled stubs and skeletons that apply a wide range of time and space optimizations. TAO 1.4a’s IDL compiler now supports both OMG IDL 2.x and 3.x features, including the latest object-by-value features.
• TAO can be configured to use non-multiplexed connections that avoid priority inversion and behave predictably when used with multi-rate distributed real-time and embedded applications. It can also be configured to use multiplexed connections, which make it more scalable when run in large-scale Internet or enterprise application environments.

• TAO’s pluggable protocols framework allows for the support of a wide range of transport mechanisms, such as standard TCP/IP protocols, UDP, UNIX-domain sockets, SSL, shared memory, and VME. This framework is also important to users developing end-to-end fault tolerant systems, where network reliability may require special purpose protocols, such as the Stream Control Transmission Protocol (SCTP), which is also supported by TAO. TAO 1.4a also supports the HTTP Tunneling Inter-ORB Protocol (HTIOP) that layers GIOP messages over HTTP packets and allows inter-ORB communication across a firewall.

• TAO’s Real-time Event Service, Notification Service, and Real-time CORBA Scheduling Service integrate the capabilities of the TAO ORB described above. These services form the basis for next-generation real-time applications for many research and commercial projects. Interestingly, these requirements are also appearing in service-sensitive application frameworks as businesses recognize the key role of middleware in ensuring predictable behavior in their major systems. TAO’s Notification Service has been overhauled and improved significantly for the OCI TAO 1.4a release.

The OCI TAO 1.4a release provides all these optimizations within the standard CORBA 2.x and 3.x (CCM) reference models. The OCI TAO 1.4a documentation set contains over 1,500 pages of text, examples, tables, and figures that explain the strategies and tactics for applying these and many other TAO, CIAO, and TAO DDS features and services to address your middleware and application needs. The size of the OCI TAO documentation set is testimony to the inherent value of their product, which can positively impact your project schedule and software development costs. The OCI TAO documentation set, along with OCI’s website dedicated to TAO (http://www.theaceorb.com) are also becoming a community repository of best practices for developing effective distributed real-time and embedded applications.

Now that TAO, CIAO, and TAO DDS have matured into widely used stable standards-based, open-source middleware platforms that are supported commercially, the future plans of my research group at Vanderbilt University’s Institute for Software Integrated Systems (ISIS) will focus on the following activities:

• Continuing to improve the quality, performance, predictability, and footprint of ACE, TAO, and CIAO. There are a number of ongoing efforts to optimize many aspects of performance, as well as to make its footprint much smaller and more configurable. The
memory footprint of the 1.5 release of TAO will be significantly smaller (i.e., less than half as large) and the build times for TAO and applications that use TAO will be significantly faster.

- Adding support for emerging standard OMG services and features. TAO 1.4a is the first release to contain support for the OMG Lightweight CCM, Component Deployment and Configuration, and Real-time Notification Service, and DDS specifications. The 1.5 release of TAO will contain support for Real-time CCM, which combines Lightweight CCM with Real-time CORBA.

- Porting TAO to new platforms. TAO 1.4a continues to support a broad range of platforms, and now covers all mainstream UNIX and Windows platforms, as well as many real-time and embedded operating systems. The portability of TAO is testimony to the ACE abstraction layer and the recognition that special-purpose operating systems will be here for a long time. Hardly a month passes without a new OS or a variant being added to satisfy a specific market need.

- Combining CIAO and TAO DDS with model-driven development (MDD) tools that enable developers to (1) model and analyze application functionality and QoS requirements as a platform-independent model and (2) synthesize deployment metadata, which is a platform-specific model, required to deliver end-to-end QoS.

- Adding tools that enable design, test, and management of complex distributed real-time and embedded applications and systems. We cannot manage, or improve, what we cannot measure. Our user base must also now help us evolve tools which can keep pace with the TAO evolution and mitigate the risk of deploying such systems. Users should look to folding new TAO features and TAO tools into their project plans so as to sustain the momentum of this project.

All of these enhancements will be integrated into future products and documentation sets supported by OCI.

I have long felt that the open-source community is fortunate that OCI has committed itself to supporting TAO, CIAO, and TAO DDS using an open-source business model. Many users would not have been able to go to their management without the assurance of commercial quality products and services. This support comes at a time when open-source software is achieving critical mass, and users are not just accepting it, but are recognizing its importance in the mix of software development models. No longer are choices restricted to “off the shelf” or custom development. There is a third way now. A way in which users participate heavily in the open source process and sponsor the fulfillment of their unmet needs as “feature additions.” Over the last five to six years of supporting open-source middleware, many of
OCI's contributions have been at the client's behest for a feature or port in a timely manner. By adding JacORB, an open-source Java ORB, to its supported products, OCI has met a need that many TAO users felt for a complementary Java solution.

In closing, it is important to recognize the extent to which the success of TAO, CIAO, and TAO DDS has benefited from OCI’s open-source development model. We're proud to have so many bright staff, students, OCI engineers, and members of the open-source community working with us over the years. As you work with TAO, CIAO, and TAO DDS, please feel free to experiment with, dissect, repair, and improve it. We accept bug reports, appreciate bug fixes/enhancements, and strive to integrate correct bug fixes quickly using our online problem tracking system. We look forward to seeing your name in subsequent releases of our software!

Douglas C. Schmidt
Preface

What Is TAO?

TAO (The ACE ORB) is an open source, advanced, CORBA-compliant, real-time Object Request Broker (ORB). Its research-guided and industry-driven middleware architecture is designed to meet the stringent Quality of Service (QoS) requirements of real-time applications. This focus on QoS requirements has resulted in TAO’s superior end-to-end predictability, efficiency, and scalable performance. TAO has been built with components from the ACE (ADAPTIVE Communication Environment) framework allowing for a highly extensible architecture. Although TAO was designed to meet the demanding requirements of real-time applications, it is also well-suited for general-purpose CORBA applications that do not have stringent QoS requirements.

Licensing Terms

TAO is made available under the open source software model. The source code may be freely downloaded and is open for inspection, review, comment,
and improvement. Copies may be freely installed across all your systems and those of your customers. There is no charge for development or run-time licenses. The source code is designed to be compiled, and used, across a wide variety of hardware and operating system architectures. You may modify it for your own needs, within the terms of the license agreements. You must not copyright ACE/TAO software. For details of the licensing terms, as specified by the Center for Distributed Object Computing at Washington University in St. Louis, please refer to “Detailed Licensing Terms” on page xxxix.

TAO also utilizes, and is distributed with, two other open source software products; GPERF and MPC. The open source licenses for these products are similar to that of ACE and TAO. Detailed licensing terms for GPERF are found on page xli. Detailed licensing terms for MPC are found on page xliii.

TAO utilizes two software products obtained/derived from Sun Microsystems. The first product implements the OMG’s Internet Inter-ORB Protocol (IIOP). You may copy, modify, distribute, or sublicense the licensed product without charge, as part of a product or software program developed by you, so long as you preserve the interoperability specified by the OMG’s IIOP.

The second Sun Microsystems product implements an OMG Interface Definition Language (IDL) compiler front-end. You may also include this product freely in any distribution, and may modify it, as long as you do not remove functionality.

In both cases, you must not use the Sun Microsystems name, logo, or copyrighted material in any subsequent distribution or promotion of your product. In addition, you must include the Sun Microsystems licensing terms, which can be found in their entirety on page xli and page xliii.

TAO is also distributed with another open source product, TAO DDS. TAO DDS is an implementation of the Data-Centric Publish-Subscribe (DCPS) layer of the OMG Data Distribution Service for Real-Time Systems (OMG Document formal/04-12-02). The open source license for this product is similar to that of ACE and TAO. Detailed licensing terms for TAO DDS are found on page xlv.

TAO is open source and the development group welcomes code contributions. Active participation by users ensures a robust implementation. Before you send code, please refer to the terms and conditions relating to software submissions on the DOC group’s TAO web site, accessible via
Incorporation of your code into TAO means that it is now “open.” The ACE/TAO copyright and terms protect the user community from legal infringement or violation.

About This Guide

This Developer’s Guide is the fifth edition and corresponds to OCI’s Distribution of TAO Version 1.4a. It extends the previous edition which corresponded to OCI’s Distribution of TAO Version 1.3a. The publication and release of this edition does not mean the previous edition is obsolete. Much of the information in the previous edition still applies to the TAO 1.4a release. However, some features/options described in the previous edition are deprecated in TAO 1.4a and are so noted in the text.

This guide focuses on the aspects of TAO that make it unique from other ORBs. It is not meant to be a comprehensive CORBA developer’s guide. Please refer to Advanced CORBA Programming with C++ by Michi Henning and Steve Vinoski or Pure CORBA by Fintan Bolton for a more complete treatment of general-purpose CORBA programming topics.

Highlights of the TAO 1.4a Release

OCI’s Distribution of TAO Version 1.4a includes many new features and improvements over the previous release. This section highlights some of the more important and visible changes and describes how they may impact your existing TAO applications.

CORBA Compliance
See Appendix I for details of TAO’s compliance with various OMG specifications. Briefly:

- **CORBA Core**—TAO implements most of the CORBA 3.0 Core specification.
- **Wide Character Support**—TAO’s support for wide characters and codeset conversion is improved.
- **Reduced Footprint**—The IDL compiler’s client- and server-side generated code has been refactored, allowing a large reduction in the amount of
generated code. This refactoring can result in major reductions in the size of some TAO applications.

- **Value Types**—TAO’s support for value types has been improved in this release.

- **Attribute Exceptions**—Exceptions can now be specified for attributes in IDL interfaces, and these exceptions can be raised from the attributes’ corresponding get and set operations.

- **CORBA Component Model (CCM) Support**—TAO 1.4a now includes the Component-Integrated ACE ORB (CIAO), a CCM implementation. CIAO allows for the development and deployment of components using TAO. See Chapter 32 for details on using CIAO.

### Configuration and Platform Support

- **Make Project Creator (MPC)**—Although MPC was used in TAO 1.3a, its features and usage are much improved in TAO 1.4a. While still being able to generate build tool specific project files from a generic mpc file, MPC now supports improved customization, robustness, and performance. See Chapter 4 for full MPC documentation.

- **Ports**
  - TAO 1.4a includes support for gcc 4.0, Sun CC 5.5/5.6, and Microsoft Visual C++ 8.0.

- **Libraries**—TAO’s features and services have been further divided into additional libraries so applications only need to link libraries with the features they need. This should improve application memory footprint and library load times.

### Pluggable Protocols

- **SCTP Inter-ORB Protocol (SCIOP)**—TAO now supports this new OMG-defined protocol that uses GIOP messaging over SCTP (Stream Control Transmission Protocol).

- **HTTP Tunneling Inter-ORB Protocol (HTIOP)**—TAO now supports this new TAO-specific protocol that allows tunneling of GIOP messages through HTTP.
CORBA Services

- **Event**—Typed event channels are now supported.
- **SSLIOP**—Support for the SSLIOP protocol has improved in a number of areas including bi-directional GIOP support and interoperability.
- **Security**—TAO now supports Common Secure Interoperability version 2 (CSIv2).
- **Fault Tolerance**—TAO’s Fault Tolerant CORBA services have been enhanced with additional capabilities.
- **Notification**—Event and connection reliability properties are now supported.
- **Implementation Repository**—This service has been made much more robust, especially the per-client activation mode.
- **Data Distribution Service (DDS)**—TAO 1.4a includes an initial implementation of this service that allows for efficient distribution of high volumes of data.

Important Bug Fixes

- Many other bug fixes or work-arounds appear in this release. See $ACE_ROOT/OCIRelaseNotes.html for details.

Structure of the Guide

Part 1, “Introduction to TAO Programming”

This section discusses the design goals, development history, and architecture of TAO. It also describes TAO’s support for various aspects of the OMG CORBA specifications and discusses TAO’s extensions to these specifications to improve predictability and performance. It addresses how to obtain, build, and install the TAO source code distribution. (Detailed instructions for building in specific environments are presented in the appendices.) This section will also help you quickly get started writing and building applications with TAO. Getting started with TAO on different platforms is simpler than in the past with the help of MPC, which is also described in this section. We also introduce a simple *Messenger* example that is used throughout the guide to illustrate features of TAO as they are discussed.
Part 2, “Features of TAO”
This section describes several features of TAO, including: TAO’s IDL compiler; dealing with errors and exceptions; TAO’s implementation of the CORBA Messaging specification, including Asynchronous Method Invocation (AMI); TAO’s Asynchronous Method Handling (AMH) feature; Real-Time CORBA; Portable Interceptors, Value Types, and Smart Proxies; using local objects; using TAO’s IOR Table feature; using and developing Pluggable Protocols for TAO; and multithreading with TAO.

Part 3, “Run-time Configuration of TAO”
The role of the ACE Service Configurator in configuring TAO at run time is discussed in detail in this section. This section describes several initialization options and environment variables for configuring the ORB and fully describes configuration options for TAO’s internal resource and strategy factories.

Part 4, “TAO Services”
This section describes the various services that TAO offers to CORBA applications. These services include some of the standard CORBA services (e.g., Naming, Events, Notification, Interface Repository, Implementation Repository, Security, Data Distribution), as well as the TAO-specific Real-Time Event Service.

Part 5, “CIAO”
This section describes TAO’s implementation of the CORBA Component Model (CCM), called the Component-Integrated ACE ORB (CIAO). It includes information on specifying and building components as well as deploying components and their applications.

Part 6, “Appendices”
This section includes appendices describing how to configure, build, test, and use TAO, including detailed information for specific operating environments. Also included is an appendix that details TAO’s level of compliance with particular OMG specifications, and another that lists over one thousand contributors to TAO. Following the appendices is a list of important references.
Conventions

This guide uses the following conventions:

<table>
<thead>
<tr>
<th>Text Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed pitch text</td>
<td>Indicates example code or information a user would enter using a keyboard.</td>
</tr>
<tr>
<td><strong>Bold fixed pitch text</strong></td>
<td>Indicates example code that has been modified from a previous example or text appearing in a menu or dialog box.</td>
</tr>
<tr>
<td><em>Italic text</em></td>
<td>Indicates a point of emphasis.</td>
</tr>
<tr>
<td>...</td>
<td>A horizontal ellipsis indicates that the statement is omitting text.</td>
</tr>
<tr>
<td>.</td>
<td>A vertical ellipsis indicates that a segment of code is omitted from the example.</td>
</tr>
</tbody>
</table>

Coding Examples

Throughout this guide, we illustrate topics with coding examples. The examples in this guide are intended for illustration purposes and should not be considered to be “production-ready” code. In particular, error handling is sometimes kept to a minimum to help the reader focus on the particular feature or technique that is being presented in the example. The source code for all these examples is available as part of the ACE and TAO source code distribution in the `$TAO_ROOT/DevGuideExamples` directory. (Source code for the DDS examples is in `$DDS_ROOT/DevGuideExamples`.) The example files are arranged in subdirectories by chapter name. MPC files are provided with all the examples for generating build-tool specific files, such as GNU Makefiles or Visual C++ project and solution files. See `$TAO_ROOT/DevGuideExamples/readme.txt` for instructions on building the examples. A Perl script named `run_test.pl` is provided with each example so you can easily run it.

OMG Specification References

Throughout this guide, we refer to various specifications published by the Object Management Group (OMG). These references take the form `group/number` where `group` represents the OMG working group responsible
for developing the specification, or the keyword formal if the specification has been formally adopted, and number represents the year, month, and serial number within the month the specification was released. For example, the OMG CORBA 3.0 specification is referenced as formal/02-12-06.

You can download any referenced OMG specification directly from the OMG web site by prepending <http://www.omg.org/cgi-bin/doc?> to the specification’s reference. Thus, the specification formal/02-12-06 becomes <http://www.omg.org/cgi-bin/doc?formal/02-12-06>. Providing this destination to a web browser should take you to a site from which you can download the referenced specification document.

**Additional Documents**

In several places throughout the text, we refer to information found in the following books:


The above books provide extensive coverage of general-purpose CORBA programming topics and techniques that are not covered in this book. We strongly recommend that you obtain copies of these books if you do not already have access to them.

OCI will continue to produce documentation for TAO. In addition, we will publish any corrections or errata to the existing documentation on the OCI web site at <http://www.theaceorb.com/references/> as necessary.

Finally, be sure to visit the TAO Frequently Asked Questions (FAQ) pages at <http://www.theaceorb.com/faq/>.

**Product Version Numbering Scheme**

Version numbers for *OCI’s Distribution of TAO* are different from those of the code base maintained by the “DOC group,” even though *OCI’s Distribution of TAO* is derived from that code base. For example, TAO 1.4a is based on the
TAO 1.4.3 beta kit distributed by the DOC group with patches carefully applied to fix specific problems.

Also, note that neither distribution’s version has any relationship to OMG specification version numbers. Neither group could ever hope to keep up with the other.

See “What's the relationship between OCI's TAO and DOC's TAO?” in the OCI TAO FAQ at <http://www.theaceorb.com/faq/> for more information about the relationship between OCI’s Distribution of TAO and the DOC group’s distribution.

Note Check <http://www.theaceorb.com/references/> to find source code patches to OCI’s distributions of TAO.

Note Mixing OCI patches and DOC group patches is untested and unlikely to work correctly. Neither group supports this configuration.

Supported Platforms

TAO has been ported to a wide variety of platforms, operating systems, and C++ compilers. We continually update TAO to support additional platforms. Please visit <http://www.theaceorb.com> for the most recent platform support information.

Customer Support

Enterprises are discovering that it takes considerable experience, knowledge, and money to design and build a complex distributed application that is robust and scalable. OCI can help you successfully architect and deliver your solution by drawing on the experience of seasoned architects who have extensive experience in today's middleware technologies and who understand how to leverage the power of CORBA.

Our service areas include systems architecture, large-scale distributed application architecture, and object oriented design and development. We
excel in technologies such as CORBA (ACE+TAO and JacORB), J2EE, XML and wireless.

Support offerings for TAO include:

- Consulting services to aid in the design of extensible, scalable, and robust CORBA solutions, including the validation of domain-specific approaches, service selection, ORB customization and extension, and migrating your applications to TAO or JacORB from other ORBs.
- 24x7 support that guarantees the highest response level for your production-level systems.
- On-demand service agreement for identification and assessment of minor bugs and issues that may arise during the development and deployment of CORBA-based solutions.

Our architects have specific and extensive domain expertise in security, telecommunications, defense, financial, and other real-time distributed applications.

We can provide professionals who can assist you on short-term engagements, such as architecture and design review, rapid prototyping, troubleshooting, and debugging. Alternatively, for larger engagements, we can provide mentors, architects, and programmers to work alongside your team, providing assistance and thought leadership throughout the life cycle of the project.

Contact us at +1.314.579.0066 or <sales@ociweb.com> for more information.

Object Technology Training

OCI provides a rich program of more than 50 well-focused courses designed to give developers a solid foundation in a variety of technical topics, such as Object Oriented Analysis and Design, C++ Programming, Java Programming, Distributed Computing Technologies, Patterns, XML, UNIX/Linux, and Wireless technology. Our courses clearly explain major concepts and techniques, and demonstrate, through hands-on exercises, how they map to real-world applications.
On-Site Classes

We can provide the following courses at your company’s facility, integrating them seamlessly with other employee development programs. For more information about these or other courses in the OCI curriculum, visit our course catalog on-line at <http://www.ociweb.com/education/>.

**Introduction to CORBA**
In this one-day course, you will learn the benefits of distributed object computing; the role CORBA plays in developing distributed applications; when and where to apply CORBA; and future development trends in CORBA.

**CORBA Programming with C++**
In this hands-on, four-day course, you will learn: the role CORBA plays in developing distributed applications; the OMG’s Object Management Architecture; how to write CORBA clients and servers in C++; how to use CORBA services such as Naming and Events; using CORBA exceptions; and basic and advanced features of the Portable Object Adapter (POA). This course also covers the specification of interfaces using OMG Interface Definition Language (IDL) and details of the OMG IDL-to-C++ language mapping, and provides hands-on practice in developing CORBA clients and servers in C++ (using TAO).

**CORBA Programming with Java**
In this hands-on, four-day course, you will learn: the role CORBA plays in developing distributed applications; the OMG’s Object Management Architecture; how to write CORBA clients and servers in Java; how to use the CORBA Naming Service; and basic and advanced features of the Portable Object Adapter (POA). This course also covers the specification of interfaces using OMG Interface Definition Language (IDL) and details of the OMG IDL-to-Java language mapping, and provides hands-on practice in developing CORBA clients and servers in Java (using JacORB).
Advanced CORBA Programming Using TAO
In this intensive, hands-on, four-day course, you will learn: several advanced CORBA concepts and techniques and how they are supported by TAO; how to configure TAO components for performance and space optimizations; and how to use TAO’s various concurrency models to meet your application’s end-to-end QoS guarantees. The course covers recent additions to the CORBA specifications and to TAO to support real-time CORBA programming, including Real-Time CORBA. It also covers TAO’s Real-Time Event Service, Notification Service, and Implementation Repository, and provides extensive hands-on practice in developing advanced TAO clients and servers in C++. This course is intended for experienced and serious CORBA/C++ programmers.

Using the ACE C++ Framework
In this hands-on, four-day course, you will learn how to implement Interprocess Communication (IPC) mechanisms using the ACE (ADAPTIVE Communication Environment) IPC Service Access Point (SAP) classes and the Acceptor/Connector pattern. The course will also show you how to use a Reactor in event demultiplexing and dispatching; how to implement thread-safe applications using the ACE thread encapsulation class categories; and how to identify appropriate ACE components to use for your specific application needs.

Object-Oriented Design Patterns and Frameworks
In this three-day course, you will learn the critical language and terminology relating to design patterns, gain an understanding of key design patterns, learn how to select the appropriate pattern to apply in a given situation, and learn how to apply patterns to construct robust applications and frameworks. The course is designed for software developers who wish to utilize advanced object-oriented design techniques and managers with a strong programming background who will be involved in the design and implementation of object-oriented software systems.

Introduction to Real-Time Systems
In this one-day course, you will learn to apply fundamental real-time concepts, techniques, and notation; compare canonical examples of real-time systems in several domains; navigate key design trade-offs in building real-time systems;
and resolve distinct sets of real-time design forces to find different real-time solutions in each domain.

**Introduction to Model-Driven Architecture**
In this one-day course, you will receive an overview of the OMG’s Model Driven Architecture (MDA). MDA is a complex, yet powerful, approach to developing extensible and maintainable systems. This lecture-only course provides an introduction to the foundational techniques and technologies underlying MDA, including meta-modeling, UML profiles, and the Meta Object Facility.

---

**Note**  
For information about training dates, contact us by phone at +1.314.579.0066, via electronic mail at training@ociweb.com, or visit our web site at <http://www.ociweb.com> to review the current course schedule.
Detailed Licensing Terms

The ACE ORB source code is copyrighted by Dr. Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University. The actual terms are reproduced below. TAO is made available by means of an open source model. TAO may be used without the payment of development license or run-time fees. The TAO source may be made available along with any added value products that utilize TAO. The acknowledgement of the use of TAO should conform to its copyright terms. You may reference the OCI version number and OCI web site as a location of the source code. OCI is an authorized distributor of TAO products and services. The use of the ACE, The ACE ORB and TAO trade or service marks is by permission of Dr. Douglas C. Schmidt.

TAO, under certain circumstances, also uses a software program called GPERF. This software was also written by Dr. Schmidt and is licensed under the terms of the Free Software Foundation’s GNU Public License (GPL). Details on this license may be found in this section. TAO also includes software from Sun Microsystems. This software is related to the IDL compiler and IIOP. This software may also be freely distributed without fees. The licensing details are also published in this section.

Please read this section carefully to understand your obligations as a user.

The following are the terms and conditions of The ACE ORB source code:

Copyright and Licensing Information for ACE(TM) and TAO(TM)

ACE(TM) and TAO(TM) are copyrighted by Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University Copyright (c) 1993-2005, all rights reserved. Since ACE+TAO are open-source, free software, you are free to use, modify, copy, and distribute--perpetually and irrevocably--the ACE+TAO source code and object code produced from the source, as well as copy and distribute modified versions of this software. You must, however, include this copyright statement along with code built using ACE+TAO.

Usage

You can use ACE+TAO in proprietary software and are under no obligation to redistribute any of your source code that is built using ACE+TAO. Note, however, that you may not do anything to the ACE+TAO code, such as copyrighting it yourself or claiming authorship of the ACE+TAO code, that will prevent ACE+TAO from being distributed freely using an open-source development model. You needn't inform anyone that you're using ACE+TAO in your software, though we encourage you to let us <doc_group@cs.wustl.edu> know so we can promote your project in the ACE+TAO success stories <http://www.cs.wustl.edu/~schmidt/ACE-users.html>.

Warranty

ACE+TAO are provided as is with no warranties of any kind, including the warranties of design, merchantability, and fitness for a particular purpose, noninfringement, or arising from
a course of dealing, usage or trade practice. Moreover, ACE+TAO are provided with no support and without any obligation on the part of Washington University, UC Irvine, Vanderbilt University, their employees, or students to assist in its use, correction, modification, or enhancement. A number of companies provide commercial support for ACE and TAO, however.

**Year 2000**
Both ACE and TAO are Y2K-compliant, as long as the underlying OS platform is Y2K-compliant.

**Liability**
Washington University, UC Irvine, Vanderbilt University, their employees, and students shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by ACE+TAO or any part thereof. Moreover, in no event will Washington University, UC Irvine, or Vanderbilt University, their employees, or students be liable for any lost revenue or profits or other special, indirect and consequential damages.

**Submissions**
The ACE and TAO web sites are maintained by the Center for Distributed Object Computing <http://www.cs.wustl.edu/~schmidt/doc-center.html> of Washington University for the development of open-source software as part of the open-source software community. By submitting comments, suggestions, code, code snippets, techniques (including that of usage), and algorithms, submitters acknowledge that they have the right to do so, that any such submissions are given freely and unreservedly, and that they waive any claims to copyright or ownership. In addition, submitters acknowledge that any such submission might become part of the copyright maintained on the overall body of code, which comprises the ACE and TAO software. By making a submission, submitter agree to these terms. Furthermore, submitters acknowledge that the incorporation or modification of such submissions is entirely at the discretion of the moderators of the open-source ACE+TAO projects or their designees.

**Trademarks**
The names ACE(TM), TAO(TM), Washington University, UC Irvine, and Vanderbilt University, may not be used to endorse or promote products or services derived from this source without express written permission from Washington University, UC Irvine, or Vanderbilt University. Further, products or services derived from this source may not be called ACE(TM) or TAO(TM), nor may the name Washington University, UC Irvine, or Vanderbilt University appear in their names, without express written permission from Washington University, UC Irvine, and Vanderbilt University.

**Contact**
If you have any suggestions, additions, comments, or questions, please let me <schmidt@cs.wustl.edu> know.

Douglas C. Schmidt <http://www.cs.wustl.edu/~schmidt/>
Copyright and Licensing Information for GPERF

GPERF is a standalone software program. GPERF generates perfect hash functions for lookups based on a set of key words when the key words are known in advance. They are called perfect hash functions because only a single access into the data structure is needed to perform a lookup. When the set of IDL operations is known in advanced TAO uses the perfect hash functions generated by GPERF to perform the operation lookup in constant time.

GPERF was originally developed by Dr. Douglas C. Schmidt. Dr. Schmidt subsequently signed the copyright over to the Free Software Foundation, causing gperf to be licensed under the GPL (GNU General Public License). The FSF still maintains that version of gperf. When perfect hashing was added as an option to TAO, gperf was selected to provide that function. It was extended and enhanced to meet the more demanding needs of TAO and a derived version was placed in the ACE application libraries. When using TAO under certain circumstances you may elect to use that version of gperf, which is part of the ACE distribution of examples and optional programs. Both the current FSF gperf and the ACE gperf are based on the original implementation. Since the ACE gperf is derived from the original GPL'ed version, it too is licensed under the GPL.

The following terms are found in the source files for gperf:

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, or visit their web site www.gnu.ai.mit.edu.

IDL Compiler Front End

This product is protected by copyright and distributed under the following license restricting its use.

The Interface Definition Language Compiler Front End (CFE) is made available for your use provided that you include this license and copyright notice on all media and documentation and the software program in which this product is incorporated in whole or part. You may copy and extend functionality (but may not remove functionality) of the Interface Definition Language CFE without charge, but you are not authorized to license or distribute it to anyone else except as part of a product or program developed by you or with the express written consent of Sun Microsystems, Inc. (“Sun”).

The names of Sun Microsystems, Inc. and any of its subsidiaries or affiliates may not be used in advertising or publicity pertaining to distribution of Interface Definition Language CFE as permitted herein.
This license is effective until terminated by Sun for failure to comply with this license. Upon termination, you shall destroy or return all code and documentation for the Interface Definition Language CFE.

INTERFACE DEFINITION LANGUAGE CFE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

INTERFACE DEFINITION LANGUAGE CFE IS PROVIDED WITH NO SUPPORT AND WITHOUT ANY OBLIGATION ON THE PART OF Sun OR ANY OF ITS SUBSIDIARIES OR AFFILIATES TO ASSIST IN ITS USE, CORRECTION, MODIFICATION OR ENHANCEMENT.

SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY INTERFACE DEFINITION LANGUAGE CFE OR ANY PART THEREOF.

IN NO EVENT WILL SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES BE LIABLE FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL DAMAGES, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

Sun, Sun Microsystems and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc.

SunSoft, Inc. 2550 Garcia Avenue Mountain View, California 94043

NOTE:
SunOS, SunSoft, Sun, Solaris, Sun Microsystems or the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc.

**IIOP Engine**

This notice applies to all files in this software distribution that were originally derived from SunSoft IIOP code (these files contain Sun Microsystems copyright notices).

COPYRIGHT AND LICENSING

Copyright 1995 Sun Microsystems, Inc. Printed in the United States of America. All Rights Reserved.

This software product (LICENSED PRODUCT), implementing the Object Management Group’s “Internet Inter-ORB Protocol”, is protected by copyright and is distributed under the following license restricting its use. Portions of LICENSED PRODUCT may be protected by one or more U.S. or foreign patents, or pending applications.
LICENSED PRODUCT is made available for your use provided that you include this license and copyright notice on all media and documentation and the software program in which this product is incorporated in whole or part.

You may copy, modify, distribute, or sublicense the LICENSED PRODUCT without charge as part of a product or software program developed by you, so long as you preserve the functionality of interoperating with the Object Management Group’s “Internet Inter-ORB Protocol” version one. However, any uses other than the foregoing uses shall require the express written consent of Sun Microsystems, Inc.

The names of Sun Microsystems, Inc. and any of its subsidiaries or affiliates may not be used in advertising or publicity pertaining to distribution of the LICENSED PRODUCT as permitted herein.

This license is effective until terminated by Sun for failure to comply with this license. Upon termination, you shall destroy or return all code and documentation for the LICENSED PRODUCT.

LICENSED PRODUCT IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

LICENSED PRODUCT IS PROVIDED WITH NO SUPPORT AND WITHOUT ANY OBLIGATION ON THE PART OF SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES TO ASSIST IN ITS USE, CORRECTION, MODIFICATION OR ENHANCEMENT.

SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY LICENSED PRODUCT OR ANY PART THEREOF.

IN NO EVENT WILL SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES BE LIABLE FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL DAMAGES, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

SunOS, SunSoft, Sun, Solaris, Sun Microsystems and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc.

SunSoft, Inc. 2550 Garcia Avenue Mountain View, California 94043

Make Project Creator (MPC)

ACE and TAO are delivered with an easy to use, open source, freely available, build environment called MPC. The following are the terms for its usage.
Copyright and Licensing Information for MPC

MPC (Licensed Product) is protected by copyright, and is distributed under the following terms.

MPC (Make, Project, and Workspace Creator) is an open-source tool developed by OCI and written in Perl. It is designed to generate a variety of tool-specific project files from a common baseline. Through the powerful combination of inheritance and defaults, MPC is able to reduce the maintenance burden normally associated with keeping multiple target platforms, their unique build tools, and inconsistent feature sets current. It is also easily extensible to support new build environments. The objective is to solve the prevalent problem of fragile build environments usually experienced by developer groups by replacing it with a singular, robust build environment and an active community of users committed to its evolution.

Since MPC is open source and free of licensing fees, you are free to use, modify, and distribute the source code, as long as you include this copyright statement.

In particular, you can use MPC to build proprietary software and are under no obligation to redistribute any of your source code that is built using MPC. Note, however, that you may not do anything to the MPC code, such as copyrighting it yourself or claiming authorship of the MPC code, that will prevent MPC from being distributed freely using an open-source development model.

Warranty

LICENSED PRODUCT IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY, AND FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

Support

LICENSED PRODUCT IS PROVIDED WITH NO SUPPORT AND WITHOUT ANY OBLIGATION ON THE PART OF OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES TO ASSIST IN ITS USE, CORRECTION, MODIFICATION, OR ENHANCEMENT.

Support may be available from OCI to users who have agreed to a support contract.

Liability

OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS, OR ANY PATENTS BY LICENSED PRODUCT OR ANY PART THEREOF.

IN NO EVENT WILL OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES BE LIABLE FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL DAMAGES, EVEN IF OCI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

MPC copyright OCI. St. Louis MO USA, 2003-2005
TAO DDS

TAO DDS (Licensed Product) is protected by copyright, and is distributed under the following terms.

TAO DDS is an open source implementation of the Object Management Group (OMG) Data Distribution Service (DDS), developed and copyrighted by Object Computing Incorporated (OCI). TAO DDS is designed to extend “The ACE ORB” (TAO) by providing an optional specialized OMG service, known as the Data Distribution Service. The OMG DDS specification is intended to be suitable for systems whose requirements include real-time, high volume, robustness, failure tolerant data distribution, and employing a publish and subscribe model. TAO DDS is another CORBA service continuing in the tradition of TAO as a fully featured implementation of CORBA with an active community of users committed to its evolution.

Since TAO DDS is open source and free of licensing fees, you are free to use, modify, and distribute the source code, as long as you include this copyright statement.

In particular, you can use TAO DDS to build proprietary software and are under no obligation to redistribute any of your source code that is built using TAO DDS. Note however, that you may not do anything to the TAO DDS code, such as copyrighting it yourself or claiming authorship of the TAO DDS code, that will prevent TAO DDS from being distributed freely using an open source development model.

Warranty

LICENSED PRODUCT IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

Support

LICENSED PRODUCT IS PROVIDED WITH NO SUPPORT AND WITHOUT ANY OBLIGATION ON THE PART OF OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES TO ASSIST IN ITS USE, CORRECTION, MODIFICATION OR ENHANCEMENT.

Support may be available from OCI to users who have agreed to a support contract.

Liability

OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY LICENSED PRODUCT OR ANY PART THEREOF.

IN NO EVENT WILL OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES BE LIABLE FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL DAMAGES, EVEN IF OCI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TAO DDS copyright OCI. St. Louis MO USA, 2005
Part 1

Introduction to TAO Programming
CHAPTER 1

Introduction

TAO (The ACE ORB) is an open source, advanced, CORBA-compliant, real-time Object Request Broker (ORB) that has been developed under the direction of Dr. Douglas C. Schmidt by members of the Distributed Object Computing (DOC) Group. The DOC Group is a distributed research consortium lead by Dr. Schmidt and consisting of the DOC group in the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, Nashville, the Center for Distributed Object Computing in the Computer Science department at Washington University, and the Laboratory for Distributed Object Computing in the Electrical Engineering and Computer Science department at the University of California, Irvine. The DOC Group also includes members of several companies and organizations all over the world, including Object Computing, Inc., Siemens ZT, University of Maryland, Remedy IT, Riverace Corporation, PrismTech, LMCO-ATL, Qualcomm, Hewlett-Packard, and Automated Trading Desk.

The purpose of the DOC group is “to support advanced research and development (R&D) on middleware and modeling tools using an open source software development model, which allows academics, developers, and end-users to participate in leading-edge R&D projects driven by the free market of ideas, requirements, and resources.”
Introduction

In addition to supporting advanced R&D projects, TAO is being used in many commercial and government distributed applications in the areas of defense, telecommunications, multimedia, finance, manufacturing, information and systems management, and many others. Such users benefit from (and contribute to) ACE and TAO’s open source model, yet demand stable, well documented, and commercially supported releases of TAO.

OCI meets the needs of this demanding TAO user community through the publication of this guide, a controlled release cycle, free source code plus inexpensive binary distributions, commercial support (including 24x7 support for organizations using TAO in their deployed applications), training for beginning through advanced users, and participation in the OMG.

To learn more, visit OCI’s web site at <http://www.ociweb.com> or <http://www.theaceorb.com>, or contact <sales@ociweb.com>.

1.1 Design Goals

The design goals of TAO include the following:

- **Well-suited for real-time environments.**
  
  TAO’s design includes real-time requirements considerations, including avoiding end-to-end priority inversion, maintaining upper bounds on latency and jitter, and providing bandwidth guarantees. In particular, TAO improves the predictability of many intra-ORB functions, such as endpoint and request demultiplexing, concurrency control, and operation dispatching. Moreover, TAO enables applications to specify their quality of service (QoS) requirements to ORB endsystems.

- **Well-suited for conventional environments.**
  
  TAO is well-suited for conventional (i.e., non real-time) environments, and may be used as a general-purpose ORB without leveraging any of its real-time features. This allows developers already familiar with CORBA to get up to speed quickly using TAO and later take advantage of its real-time QoS support as needed.

- **Exhibits high-performance characteristics.**
  
  TAO meets the stringent throughput requirements that are necessary to support performance-sensitive industrial-strength applications. TAO
1.1 Design Goals

optimizes many of the key determinants of ORB performance, including request demultiplexing and operation dispatching, presentation layer conversions, synchronization, context switching, memory management, and data copying. As a result, TAO’s performance is very competitive with lower-level networking APIs, such as sockets.

- **Complies with open standards.**
  TAO was designed to be compliant with OMG CORBA specifications. In general, the current version of TAO is compliant with the CORBA 3.0 specification. Because of its compliance with OMG specifications, TAO is interoperable with other ORBs, and TAO application code written in accordance with the OMG’s IDL to C++ language mapping is portable to other compliant ORBs. See Appendix I for complete information on TAO’s level of compliance with various specifications.

- **Configurable.**
  When used in special purpose (e.g., embedded) environments, software must often be tailored to meet the special demands imposed by that environment. TAO’s modular architecture and run-time configurability allow developers to tailor it to meet the specific needs of their application’s operating environment (e.g., to reduce the run-time memory footprint or to strategize request demultiplexing for more deterministic performance). In addition, TAO fully supports the standard CORBA policy framework that allows developers to control ORB behavior programmatically. TAO is designed to give the developer a great deal of control over the run-time environment.

- **Extensible.**
  TAO comes with default resource and strategy factories for configuring clients and servers, and default pluggable protocol factories for choosing among certain transport protocols. These default factories are designed to provide enough flexibility to meet the needs of the vast majority of applications, even for very demanding environments. However, in cases where the default factories cannot satisfy a project’s special needs, you can develop custom resource, strategy, and protocol factories that can be “plugged in” to TAO’s core framework-based architecture with no impact on application code.

  TAO also supports the OMG Portable Interceptors specification. Portable Interceptors provide hooks that are invoked at predefined points in the
request and reply paths of an operation invocation or during the generation of an IOR. Interceptors are registered with the ORB via ORB initializers. Developers can define their own code to be executed at each interception point to perform application-specific tasks such as logging, debugging, or security and authentication. See Chapter 10 for more information on Portable Interceptors.

TAO also supports Smart Proxies, which allow developers to replace the normal proxy implementations, generated by the IDL compiler, with application-specific proxies for customizing the behavior of client request invocations. See Chapter 12 for more information on Smart Proxies.

In addition, because TAO is open source, you are free to modify it in almost any way to meet your project’s unique requirements.

- Portable.

For an ORB to be an effective tool, it must be implemented on all the platforms on which it is needed. TAO is built on top of the ADAPTIVE Communication Environment (ACE) framework, which is a C++ framework that provides object-oriented abstractions to operating system, networking, and interprocess communications facilities, as well as higher level patterns that encapsulate common communications mechanisms. By utilizing ACE, the TAO source code is more easily ported to diverse target platforms than would be the case if specific APIs were utilized. Moreover, the use of ACE mitigates the least common denominator approach of typical porting solutions. Thus, TAO can be optimized for any platform.

1.2 Development History

Since the early 1990s, Dr. Schmidt has led teams conducting advanced research and development on distributed-object computing middleware using an open source software development model. The open source model is a very pragmatic way of evolving software in a rapidly changing environment. It harnesses the collective wisdom, experiences, expertise, and requirements of its most demanding users to ensure that their needs are rapidly met.

Traditionally, ORBs have implemented only best-effort service models. Many corporate and government organizations have sponsored the development of TAO because they need both standards-based middleware and the ability to meet and enforce QoS for their applications and distributed systems. These
organizations require not just classic hard real-time characteristics, but soft real-time and best-effort support, as well. A partial list of these industries and government organizations, and their applications, includes:

- **Telecommunications**—switching, network management, software-defined radio, and mobile/hand-held systems.
- **Medical**—imaging, integrated patient monitoring, and tele-medicine.
- **Aerospace and Defense**—avionics, signal processing, simulation, command and control, and training.
- **Financial Services**—trading services, portfolio analysis, real-time risk analysis, and simulation.
- **Manufacturing**—machine tools, robotics, and process control.
- **Information and Systems Management**—storage management, systems and data recovery, customer information management, asset management, capacity management, and infrastructure and application control.

In addition to requirements for real-time and high-performance systems, TAO’s sponsors require close conformance to the OMG specifications to ensure that their developers can design compliance into their baseline systems architecture. Moreover, there are often specific customer-application-generated requirements that ensure a pragmatic set of extensions to the OMG specifications. These extensions must meet the needs of real-time developers who typically demand complete control of all the system resources to guarantee success.

Original contributors to TAO’s technical architecture, strategies and techniques include senior developers from sponsoring organizations who have extensive experience with first generation ORBs, understand real-time issues, and come from diverse industry backgrounds. The result of such wide-ranging inputs is an ORB with a highly adaptable architecture, well-suited for a diverse and demanding customer base. When combined with the thousands of contributors from the ACE and TAO open source community, it is fair to say that no other ORB has had such a significant degree of participation from its users and sponsors.
1.3 Architecture of TAO

In this section, we describe the architecture of TAO. If you are new to CORBA, you should read Advanced CORBA Programming with C++, Chapter 2 (especially sections 2.4 and 2.5) before reading this section.

Figure 1-1 shows the relationships among TAO’s ORB endsystem components. An ORB endsystem is an endsystem (e.g., PC, workstation, embedded processor board) that contains one or more network interfaces, an I/O subsystem (e.g., containing the operating system’s protocol stacks like
TCP/IP), an ORB, and possibly various standard services (e.g., Naming and Event). As a developer, you will typically write the client and servant (shaded components in Figure 1-1). Components of the TAO architecture are described below.

1.3.1 **Client**

A CORBA client has two responsibilities: (1) obtain object references to CORBA objects and (2) invoke operations on them. The client is unaware of where or how the CORBA object is implemented. The only operations the client is able to invoke are those defined in the object’s interface, expressed in OMG Interface Definition Language (IDL).

1.3.2 **CORBA Object**

CORBA objects are abstract entities. Each CORBA object has a unique identity and an interface, defined in IDL. A CORBA object is associated with a concrete implementation of the interface at run time by an Object Adapter.

1.3.3 **Servant**

A servant provides a concrete implementation for a CORBA object. In object-oriented programming languages such as C++, servants are implemented as objects and live within a server process or task. Normally, you will create an implementation class for each IDL interface and your servant objects will be instances of this class. The client is completely unaware of how an interface is implemented and has no knowledge of servants. A servant is associated with a CORBA object at run time via an Object Adapter.

1.3.4 **IDL Stubs and Skeletons**

The TAO IDL compiler generates C++ stubs and skeletons from IDL interface definitions. Stubs are used on the client side to provide a strongly typed, static invocation interface (SII) that converts C++ function calls into CORBA requests, including marshaling operation parameters into a common binary representation. The generated stubs can also optimize operation invocations when the target object is collocated with (i.e., in the same address space as) the client. Skeletons provide a static skeleton interface (SSI) that demarshals the binary data back into C++ types that are meaningful to servant implementations. You will normally compile and link the generated stubs and
skeletons into your application code. See Chapter 5 for more information on TAO’s IDL compiler.

In addition to the SII and SSI model described above, TAO also supports the Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI), defined by the CORBA specification.

1.3.5 Portable Object Adapter

The Portable Object Adapter (POA) specification, introduced in CORBA 2.2, replaces the Basic Object Adapter (BOA) defined in earlier versions of the CORBA specification. An Object Adapter associates servants with CORBA object references, demultiplexes incoming requests, and dispatches these requests to servants.

TAO fully implements the POA specification, including support for multiple nested POAs per ORB, applying policies to POAs at creation time, and portability of server implementation code. In addition, TAO's POA, by default, optimizes request demultiplexing and operation dispatching, using active demultiplexing and perfect hashing. These optimizations improve the predictability of CORBA applications by ensuring constant O(1) time operation dispatches, regardless of the number of active client connections, the number of activated servants, and the number of operations defined in an IDL interface. Alternative lookup strategies are configurable, such as linear search, binary search, and dynamic hashing.

The RTPortableServer extension to the POA interface, which is part of the real-time CORBA specification, adds operations that permit the application to associate priorities with object activations and to define thread pools for operation dispatching. The RT CORBA specification also defines a system of portable priorities that can be mapped to native operating system priorities. Thus, RT CORBA provides a single, “global” priority model that simplifies system design and improves code portability and extensibility. RT CORBA provides a standard mechanism for servers to allocate, partition, and manage thread resources and control dispatching of requests onto threads according to priority, thereby helping to ensure end-to-end predictability. In addition, RT CORBA gives the developer control over the allocation and selection of communication resources via explicit binding, protocol configuration, and protocol selection. See Chapter 9 for more information on TAO's implementation of real-time CORBA.
1.3.6 ORB Core

A client ORB communicates with a server ORB to deliver client request messages and return responses, if any, to the client. On the server side, the ORB Core delivers the request to the appropriate Object Adapter and returns a reply message to the client-side ORB. ORBs also actively manage the transport-level connections that are used to transmit these request and reply messages.

The OMG defines the General Inter-ORB Protocol (GIOP) for enabling interoperable communications among disparate ORB implementations. TAO’s ORB Core supports GIOP version 1.2 (and its realization atop the TCP transport protocol, known as the Internet Inter-ORB Protocol or IIOP). In addition, TAO’s pluggable protocols framework allows GIOP messaging to operate over a wide range of transport protocols, including user-defined transports. In addition to IIOP, TAO provides alternate pluggable transport protocols, such as: UIOP, for inter-ORB communications over local IPC (or UNIX domain sockets); SHMIOP, for inter-ORB communications via shared memory; DIOP, for limited but highly-efficient inter-ORB communications using UDP; SSLIOP, for secure inter-ORB communications using Secure Sockets Layer (SSL); MIOP, for inter-ORB communications over unreliable multicast protocol; SCIOP, for inter-ORB communications over the Stream Control Transmission Protocol (SCTP); and HTIOP, which tunnels inter-ORB communications over Hypertext Transfer Protocol (HTTP). Each pluggable transport protocol must provide a protocol factory that is loaded and configured at run time. See Chapter 15 for more information on using TAO’s pluggable protocols.

In addition to the pluggable protocols framework, TAO’s ORB Core employs various strategies to configure certain aspects of the ORB’s behavior for both the client and server sides. For example, on the client side, strategies are used to affect concurrency, to determine if multiple requests are allowed to share a communication channel, to control connection management by the ORB core, and various other behaviors. On the server side, strategies are used to control how the object adapter demultiplexes requests and to control concurrency. To obtain these strategies, the ORB core uses strategy factories that are loaded and configured at run time via the ACE Service Configurator framework. See Part 3, “Run-time Configuration of TAO,” for more information on configuring the ORB’s strategy factories.
1.3.7 **ACE**

TAO is implemented atop ACE, which is infrastructure middleware that implements the core concurrency and distribution patterns for communication software. ACE provides reusable C++ wrapper façades and framework components that support the QoS requirements of high-performance, real-time applications. ACE is a highly portable, multiplatform framework that spans both real-time and general-purpose operating systems.

1.4 **CORBA Compliance**

It is not necessary to use TAO as a real-time ORB. In fact, TAO provides out-of-the-box standard CORBA conformance. TAO was designed to be compliant with OMG CORBA specifications, as summarized below and as detailed in Appendix I.

- TAO is mainly compliant with the OMG CORBA 3.0.3 specification (OMG Document formal/04-03-12).
- TAO implements the *minimumCORBA* specification (OMG Document formal/02-08-01).
- TAO implements the real-time CORBA 1.2 specification (OMG Document formal/05-01-04).
- TAO is compliant with the CORBA C++ Language Mapping specification, version 1.1 (OMG Document formal/03-06-03).
- TAO complies with the Internet Inter-ORB Protocol (IIOP) specification, version 1.2, including support for bi-directional communications over a single connection. Therefore, TAO can interoperate seamlessly with other ORBs that use the standard IIOP (including ORBs that use IIOP versions 1.0 and 1.1). TAO does not technically support IIOP 1.3, but does support the component-related features of this version through its IIOP 1.2 implementation.
- TAO supports the static invocation interface (SII) and static skeleton interface (SSI), as well as the dynamic invocation interface (DII) and dynamic skeleton interface (DSI) models.
- TAO fully implements the Portable Object Adapter (POA) specification, including advanced POA features, such as servant managers and adapter activators.
1.4 CORBA Compliance

- TAO provides many of the standard CORBA services, as follows:
  - **Audio/Video Streaming Service**—implements the Control and Management of Audio/Video Streams specification.
  - **Concurrency Control Service**—allows objects in a distributed system to acquire and release locks.
  - **Data Distribution Service**—Allows for the control of high performance, typed, data streams using a publish/subscribe paradigm.
  - **Event Service**—decouples communication between objects by providing an asynchronous supplier/consumer style of event propagation among objects.
  - **Interface Repository**—maintains a repository of information about IDL interfaces and types and provides lookup capabilities to clients.
  - **Life Cycle Service**—provides a standard means to locate, move, copy, and remove objects.
  - **Load Balancing Service**—provides random, round-robin, and least-loaded load balancing strategies to forward requests to registered replica services.
  - **Logging Service**—provides event-based logging and log-record query capabilities.
  - **Naming Service**—maps names to object references, organized in a hierarchy.
  - **Notification Service**—extends the CORBA Event Service with the addition of features such as event filtering and structured events.
  - **Property Service**—allows applications to associate properties with objects dynamically.
  - **Security Service**—provides a comprehensive treatment of security as it relates to distributed object systems and applications.
  - **Time Service**—provides globally-synchronized time to distributed objects.
  - **Trading Service**—maps properties to object references and provides constraint-based object lookup capabilities to clients.
In addition, TAO provides the following additional service that demonstrates TAO's capabilities in various real-time environments:

- **TAO Real-Time Event Service**—augments the standard CORBA Event Service model by providing source- and type-based event filtering, event correlation, priority-based dispatching, and event channel federation.

See Part 4, “TAO Services,” for more information on the various services implemented by TAO.

### 1.5 High Performance and Real-Time Support

Historically, CORBA has supported only “best-effort” quality of service to applications. Developers with stringent QoS or performance requirements could not rely on CORBA to provide the level of performance or predictability they needed.

TAO was designed from the beginning with support for real-time and other demanding applications in mind. Because this kind of support was lacking from the CORBA specifications, TAO supplied extensions to the CORBA specifications to support applications that required higher performance, real-time determinism, and end-to-end priority propagation.

Because the CORBA specification now supports more demanding applications, ORB implementations can now provide much greater QoS and performance guarantees without sacrificing CORBA compliance. TAO is in the forefront of support for these latest aspects of the CORBA specification as follows:

- TAO implements the CORBA policy framework as defined by the CORBA Messaging specification and supports the creation of policies for controlling request/reply timeouts, synchronization scope for oneway requests, support for bi-directional GIOP communications, and other aspects of inter-ORB communications.

- TAO implements the real-time CORBA specification, including the real-time ORB and real-time PortableServer features, such as portable priorities, client-propagated and server-declared priority models, RT CORBA threadpools, and priority-banded connections. In addition, TAO provides an implementation of RT CORBA dynamic scheduling.
1.6 Relationship Between ACE and TAO

In addition, TAO provides the following extensions to the CORBA specifications to support specific application needs:

- TAO’s ORB Core provides an efficient and predictable communication infrastructure for high-performance and real-time applications. It provides a range of client and server concurrency models.
- TAO’s ORB Core supports nested upcalls with several of its concurrency models.
- TAO’s implementation of RT CORBA thread pools with lanes provides a reactor-per-lane configuration that requires no context switches throughout the life of an upcall, thereby greatly decreasing the likelihood of priority inversions.
- TAO’s ORB Core allows custom transport protocols to be plugged into the ORB without affecting standard CORBA application programming interfaces.
- Some custom transport protocols supported by TAO improve request transmission performance relative to the standard IIOP protocol under certain conditions.
- TAO’s implementation of the POA and generated skeletons are designed using patterns that provide an extensible and highly optimized set of request demultiplexing and operation dispatching strategies, such as perfect hashing and active demultiplexing. These strategies allow for constant-time lookup of nested POAs and servants, based on object keys, and operation names contained in CORBA requests.

1.6 Relationship Between ACE and TAO

Many components in TAO, such as its ORB Core, POA, and generated stubs and skeletons, are based on patterns and components provided by the ACE framework. Key patterns used in TAO include the Acceptor, Connector, Reactor, Active Object, Half-Sync/Half-Async, Service Configurator, Thread-Specific Storage, Strategy, Proxy, Adapter, Bridge, and Abstract Factory.

To improve portability, TAO uses ACE’s high-performance, small-footprint operating system adaptation layer for all operating system access, rather than
invoking non-portable system calls directly. Because ACE supports numerous operating systems, porting TAO to a new platform is simplified considerably.

For more on these patterns and concepts, see the following references, found in the References section near the end of this guide.

- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (GoF).
- *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (POSA2), by Doug Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann.
- *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns* (C++NPv1), by Doug Schmidt and Steve Huston.

Also, for more information on ACE, visit the ACE home page via <http://www.theaceorb.com/references/>.
CHAPTER 2

Building ACE and TAO

2.1 Introduction

This chapter describes where to obtain ACE and TAO distributions, what resources they require on your machine, what choices you will have to make concerning the options available in the package, how to install the distribution on your computer, and if you are not using a pre-built distribution, what you will need to do to build and configure your ACE and TAO system. The actual instructions for building ACE and TAO are in various appendices. See Appendix B for more information.

2.2 Where to Get ACE and TAO

The easiest way to get started with ACE and TAO is to install a pre-built version from the *OCI’s Distribution of TAO* CD. The CD can be ordered directly from OCI (see page v for contact information) or from the web site at <http://www.theaceorb.com>. 
Building ACE and TAO

The CD offers pre-built ACE and TAO installations for several versions of UNIX and UNIX-like systems as well as for Windows systems. Refer to the web site for a detailed listing of included platforms.

The ACE and TAO distribution source code is available for downloading from OCI’s TAO web site: <http://www.theaceorb.com>.

Finally, the link to the ACE and TAO source code repository and beta kits can be found at <http://www.theaceorb.com/references/>.

2.3 System Requirements

Since ACE and TAO are used to develop software, you will need a working C++ compiler. The GNU g++ compiler is supported for most platforms, and as of this writing, g++ version 3.4 is recommended. Also, the platform vendor’s compiler is usually supported, as are certain third-party compilers. In all cases other than a Windows platform, the GNU Make program must be used. The vendor-provided linker is generally used, but the GNU linker can also be used. In very rare cases, the GNU assembler must be used. To run MPC or any of the automated test scripts, you must have Perl 5.6 or later. A complete list of the tools used to build ACE and TAO on various platforms, including version numbers and patch levels, is available in the release notes. The release notes are included with the ACE and TAO source code distribution in the file ACE_wrappers/OCIReleaseNotes.html.

How fast a processor and how much memory you need are somewhat a matter of preference and your patience for waiting on files to compile. Generally, you will want at least a 500 MHz processor and at least 256 MB RAM.

The amount of disk space you need will depend heavily on the machine, the compiler, and the build flags you choose. A production system should fit in 100 MB or less of disk space, after all the intermediary files are removed. You may need much less for a minimal build. Conversely, a full build with all debugging information and other features enabled, plus intermediary and temporary files could require over 2.0 GB of disk space.
2.4 **Make Project Creator (MPC)**

MPC is now the only way to work with ACE and TAO. It can be used to generate build files for use with various tools including GNU Make, Microsoft Visual C++, Borland Make, and Microsoft NMake. MPC eliminates the need to maintain separate build files for these various build environments, and allows easy customization of build flags on all platforms. OCI releases and DOC group beta kits both contain GNU makefiles and Visual C++ project files that were generated with MPC. You are free to use these files to build ACE/TAO but many users generate their own custom build files using MPC. For more information on how to use MPC, see Chapter 4.

2.5 **Configuring Builds**

Your build of ACE/TAO can be configured or customized in a number of ways:

- Optional features can be explicitly configured to build. These features are typically disabled because they have external dependencies on other software libraries or products. An example is Secure Sockets Layer (SSL) support. When TAO is configured to build with SSL support, it knows that the OpenSSL library is present and it should compile in support for using this transport layer.

- Normally enabled TAO features can be disabled for specific builds. This is often for memory footprint reasons or maybe to enforce project policies against using that feature. An example is Portable Interceptors support. By default, TAO supports the interceptor interfaces, but the build can be customized to disable this feature which reduces the footprint and reduces processing overhead.

- The build process itself can be customized in a number of ways, typically through compiler options. Examples of these types of customizations include enabling/disabling support for generating debugging symbols, multithreading, and function inlining.

There are three mechanisms for specifying the above configurations in your build:

- MPC features can be configured in the `default.features` file.
• GNU Make build flags can be set in the `platform_macros.GNU` file (and via other related mechanisms). This mechanism is not available with other build systems such as Microsoft Visual Studio.

• C++ macros can be set in the `config.h` file.

Some options must be set via a specific mechanism and others can be specified at multiple levels. For example, setting `ssl=1` in the `default.features` file instructs MPC to include SSL-dependent projects in the build files generated. Appendix A discusses the full details on configuring ACE/TAO builds and some of the options available via each mechanism.
CHAPTER 3

Getting Started

3.1 Introduction

This chapter guides you through the process of building and running a simple client/server application using TAO. You should already have TAO installed (from CD-ROM) or built (from source code) on your system. If not, see Chapter 2. If you are new to CORBA, you may find it helpful to read Chapter 3 of *Advanced CORBA Programming with C++* before proceeding.

TAO 1.4a uses a tool that makes using TAO essentially identical on all platforms. MakeProjectCreator (MPC) is capable of generating build files for each platform from simple text data files. So, whether you are getting started with TAO on Linux, Windows, Solaris, or one of the other many platforms supported by TAO, the steps are essentially the same.

3.1.1 Road Map

In this chapter, you will learn how to:

- Set up your environment for using TAO (see 3.2).
- Develop a simple server and client using TAO (see 3.3).
Full source code for the example presented in this chapter can be found in the TAO 1.4a source code distribution in the directory

3.2 Setting Up Your Environment

Certain environment variables are required during the compilation and run-time phases of TAO applications. These environment variables are presented here. If you built TAO yourself, these variables are probably already set and you may skip this section. The environment variables are shown first using UNIX syntax, with Windows syntax shown in parentheses.

- **ACE_ROOT**
  
  The base directory where you installed ACE and TAO, such as
  `/usr/local/ACE_wrappers` (`C:\ACE_wrappers`).

- **TAO_ROOT**
  
  The base path for all TAO-related code, normally `$ACE_ROOT/TAO`
  (`%ACE_ROOT%\TAO`).

- **PATH**
  
  Scripts and executables for TAO will be installed in `$ACE_ROOT/bin`
  (`%ACE_ROOT%\bin`). You should add this location to your **PATH**
  environment variable.

- **Library path**
  
  All required libraries will be installed in `$ACE_ROOT/lib`
  (`%ACE_ROOT%\lib`). You should add this location to your **LD_LIBRARY_PATH**
  environment variable or its equivalent. (On Windows, add this directory to your **PATH**
  so DLLs can be located at run time.)

3.3 A Simple Example

In this section, we guide you step-by-step through the creation of a simple TAO example. We create our IDL files, implement our servants, create client and server applications, generate build files, build, and run the application.
Our example consists of a server called MessengerServer that implements a simple Messenger interface, plus a client called MessengerClient that accesses and uses a Messenger CORBA object that the MessengerServer provides. Imagine that a full implementation of the MessengerServer might send e-mail, access a pager, or even make a phone call using voice synthesizer technology. To keep our example simple, we just write the client’s message to standard output. In later chapters, we will expand on this example to illustrate various TAO and CORBA features.

Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/GettingStarted.

3.3.1 Create a Workspace
First, create a working directory for our example. We will place all of our code in a single directory for this example, but in larger projects you may use a different directory structure. For example, you may wish to separate code for libraries, servers, and clients into separate subdirectories.

```bash
mkdir Messenger
cd Messenger
```

3.3.2 Messenger Interface Definition Language (IDL) File
Create a new file called Messenger.idl to contain the interface definition for our simple Messenger. This interface simply defines an operation that we will use to send text messages between a client and server. A reply may be returned in the last parameter, and the return value indicates whether the message was accepted.

```idl
interface Messenger
{
    boolean send_message(in string user_name,
                          in string subject,
                          inout string message);
};
```

3.3.2.1 Run the IDL Compiler
The IDL compiler (tao_idl) generates stub and skeleton code from the IDL interface definitions contained in Messenger.idl. Details about using the IDL compiler are found in Chapter 5. We use the -GI option to cause
tao_idl to generate _starter_ implementation (servant) files. We then modify the generated starter code for our actual implementation. Using the -GI option to automatically generate starter code is a convenient way to make sure our implementation class function signatures are correct.

    tao_idl -GI Messenger.idl

After running the IDL compiler as shown, our starter implementation class for the Messenger interface will be in files named MessengerI.*. Client-side stubs will be in files named MessengerC.* and server-side skeletons will be in files named MessengerS.*. Other files may also be generated, but they do not concern us for this simple example.

### 3.3.3 Create the Messenger_i Implementation Class

Normally, you will want to rename the generated starter implementation files MessengerI.h and MessengerI.cpp to Messenger_i.h and Messenger_i.cpp. That way, you will not inadvertently overwrite existing files if you run the IDL compiler with the -GI option again.

**UNIX**

    mv MessengerI.h Messenger_i.h
    mv MessengerI.cpp Messenger_i.cpp

**Windows**

    ren MessengerI.h Messenger_i.h
    ren MessengerI.cpp Messenger_i.cpp

### 3.3.3.1 C++ Header for the Messenger_i Class

Our Messenger_i implementation class inherits from the POA_Messenger skeleton class found in MessengerS.h. We have removed some comments and an unneeded constructor and destructor from the generated starter implementation files.

    #include "MessengerS.h"

    class Messenger_i : public virtual POA_Messenger
    {
    public:
        virtual CORBA::Boolean send_message {
            const char* user_name,
            const char* subject,
            char*& message)
3.3 A Simple Example

3.3.3.2 C++ Implementation of the Messenger_i Class

The file `Messenger_i.cpp` already contains much of the code we need for implementing the `Messenger_i` class. Here is the file with our additions and changes shown in bold text. Once again, we have removed the unneeded constructor, destructor, and some generated comments.

```cpp
#include "Messenger_i.h" // renamed from MessengerI.h
#include <iostream>

CORBA::Boolean Messenger_i::send_message (const char* user_name, const char* subject, char*& message) ACE_THROW_SPEC ((CORBA::SystemException))
{
    std::cout << "Message from: " << user_name << std::endl;
    std::cout << "Subject:      " << subject << std::endl;
    std::cout << "Message:      " << message << std::endl;
    CORBA::string_free(message);
    message = CORBA::string_dup("Thanks for the message.");
    return 1;
}
```

3.3.4 C++ Implementation of the MessengerServer

We next create a `MessengerServer` to give our `Messenger` object a place to live. In `main()`, we create an instance of our `Messenger_i` implementation class, activate it in the `RootPOA`, and wait for requests from clients.

Create `MessengerServer.cpp` with the following contents:

```cpp
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[])
{
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Get a reference to the RootPOA.
```
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

// Activate the POAManager.
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Create a servant.
Messenger_i servant;

// Register the servant with the RootPOA, obtain its object
// reference, stringify it, and write it to a file.
PortableServer::ObjectId_var oid = poa->activate_object(&servant);
obj = poa->id_to_reference(oid.in());
CORBA::String_var str = orb->object_to_string(obj.in());
ofstream iorFile("Messenger.ior");
iorFile << str.in() << std::endl;
iorFile.close();
std::cout << "IOR written to file Messenger.ior" << std::endl;

// Accept requests from clients.
orb->run();
orb->destroy();

return 0;
}
catch (CORBA::Exception& ex) {
  std::cerr << "MessengerServer CORBA exception: " << ex << std::endl;
}
return 1;

### 3.3.5 C++ Implementation of the MessengerClient

We complete our example by creating a `MessengerClient`, which obtains an object reference to the Messenger object and sends it a message via its `sendMessage()` operation.

Create `MessengerClient.cpp` with the following contents:

```cpp
#include "MessengerC.h"
#include <iostream>

int main(int argc, char* argv[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```
3.3 A Simple Example

// Read and destringify the Messenger object's IOR.
CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");
if( CORBA::is_nil(obj.in())) {
    std::cerr << "Could not get Messenger IOR." << std::endl;
    return 1;
}

// Narrow the IOR to a Messenger object reference.
Messenger_var messenger = Messenger::_narrow(obj.in());
if( CORBA::is_nil(messenger.in())) {
    std::cerr << "IOR was not a Messenger object reference." << std::endl;
    return 1;
}

// Send a message to the Messenger object.
CORBA::String_var msg = CORBA::string_dup("Hello!");
messenger->send_message("TAO User", "Test", msg.inout());

// Print the Messenger's reply.
std::cout << "Reply: " << msg.in() << std::endl;

return 0;
} catch (CORBA::Exception& ex) {
    std::cerr << "MessengerClient CORBA exception: " << ex << std::endl;
} return 1;
}

3.3.6 Create Build Files for the Example

Until recently, creating the necessary files for building TAO projects involved creating separate build tool files for each platform. For example, to build the above example on UNIX using GNU Make and on Windows using Visual C++ required building and maintaining both Makefiles and Visual C++ project/solution files. In such cross-platform environments, creating and maintaining different build files for different build tools was tedious and error-prone. This process has been greatly simplified with the introduction of a tool called MakeProjectCreator (MPC). With MPC, multiple build environments can now be supported very simply. All we have to do is create a simple mpc file with the information that is unique to our project. We then run MPC to generate build files for use with GNU Make (gmake), Microsoft Visual Studio 6 (msdev) or 7.1 (devenv), Microsoft nmake, Borland make and others. For more information on MPC see Chapter 4.
To support builds of our Messenger example, we create a file called GettingStarted.mpc with the following contents:

```
project(*Server): taoexe, portableserver {
    Source_Files {
        Messenger_i.cpp
        MessengerServer.cpp
    }
}

project(*Client): taoexe {
    Source_Files {
        MessengerC.cpp     // prevents implicit MessengerS.cpp
        MessengerClient.cpp
    }
}
```

The GettingStarted.mpc defines two projects, one for the server and one for the client. This mpc file relies on various settings inherited from base projects. Both projects inherit from taoexe, which provides all the necessary project attributes to build a TAO executable. Our server project also inherits from portableserver so that it can activate its servant in a POA. The projects will be named GettingStarted_Client and GettingStarted_Server, because we used the '*' wild card character in our project name declarations. The output files will be named MessengerClient and MessengerServer, because these are the names of the source files in each project that contain main(). MPC automatically detects the existence of our IDL files and implicitly adds these to our source files. In our client project we explicitly add MessengerC.cpp to the list of source files to prevent MPC from implicitly adding MessengerS.cpp, which we do not want to build into our client.

---

**Note**  
To use MPC, you must have Perl version 5.6.1 or greater.

The next step depends upon your development environment:

- **UNIX with GNU Make**

  On UNIX or UNIX-like systems, run `mwc.pl` in the project directory to generate GNU makefiles for use with the ACE+TAO make system:

  ```
  mwc.pl -type gnuace
  ```
The above command will generate the following files: GNUmakefile, GNUmakefile.GettingStarted_Client, and GNUmakefile.GettingStarted_Server for use with GNU Make.

- **Windows with Visual C++ 6**

  On Windows, using Visual C++ 6, run `mwc.pl` in the project directory to generate Visual Studio 6 workspace and project files:

  
  
  
  ```bash
  mwc.pl -type vc6
  ```

  The above command will generate the following files:

- **Windows with Visual C++ 7.1**

  On Windows, using Visual C++ 7.1, run `mwc.pl` in the project directory to generate Visual Studio .NET 2003 solution and project files:

  
  
  ```bash
  mwc.pl -type vc71
  ```

  The above command will generate the following files:

---

**Note**  
*Visual C++ 7.0 is not recommended for use with ACE and TAO.*

- **Windows with Borland Make**

  On Windows, using Borland C++ 5 and Borland Make, run `mwc.pl` in the project directory to generate Borland Makefiles:

  
  
  ```bash
  mwc.pl -type bmake
  ```

  The above command will generate the following files: Makefile, Makefile.GettingStarted_Client.bmak, and Makefile.GettingStarted_Server.bmak.
3.3.7 **Build the MessengerServer and MessengerClient**
Once the build files are generated, you can build the test applications.

Using GNU Make:

```
gmake (or make)
```

Using Visual C++ 6:

```
msdev /y3 GettingStarted.dsw /make "ALL - Win32 Debug"
```

Using Visual C++.NET 2003:

```
devenv GettingStarted.sln /build debug
```

Using Borland Make:

```
set debug=1
make
```

3.3.8 **Running the Application**
You are now ready to run the MessengerServer and MessengerClient. The server must be running before the client is started.

Run the MessengerServer in one terminal window with the following command:

```
./MessengerServer
```

Wait for the message "IOR written to file Messenger.ior", then run the MessengerClient from a different terminal window in the same directory with the following command:

```
./MessengerClient
```

You should see the following messages from the MessengerServer:

```
Message from: TAO User
Subject: Test
Message: Hello!
```
In the MessengerClient’s terminal, you should see:

    Reply: Thanks for the message.

indicating that the client has received a reply from the server. The client then exits and your normal command prompt reappears.

Note that the MessengerServer will still be running, waiting for more client requests. You can run the client again if you like. To kill the MessengerServer, just type `Ctrl-C` in its terminal window or use the `kill(1)` command to terminate it.

### 3.4 Summary

In this chapter, you have seen how to develop a simple server and client using TAO. Topics covered included: how to set up your environment for building applications that use TAO; how to set up a working directory for a simple example and the files to create therein; creating and using a simple mpc file for building the example; and running the example.

You are now ready to explore other chapters of this guide that expand on this simple example to illustrate various features of TAO and various services that can be used by TAO applications. *Have fun!*
CHAPTER 4

The Makefile, Project, and Workspace Creator (MPC)

4.1 Introduction

Maintaining multiple build tool files for a multi-platform project can be quite a challenge, especially when the project structure and platforms are constantly changing and evolving. A project may support Makefiles, Visual C++ project files, Borland Makefiles, and many others. Adding files, deleting files, changing project options or even changing the name of the target within your project will require you to expend time updating each build tool file. What you need instead is a single location to store project specific information to avoid repetitious, tedious modifications to multiple build tool files. This is where Makefile Project Creator (MPC) comes into the picture.

MPC can be used to generate build tool specific project files from a generic mpc file. The MPC project file is a collection of source files that make up a single build target. MPC uses platform specific input along with mpc files and generates build tool specific files like makefiles, Visual C++ workspace and project files, Visual Studio solution and project files, etc.

MPC provides many advantages over the build tool files it replaces. It provides mechanisms for minimizing maintenance of project build files. It
does this through support for project inheritance and defaults for all aspects of a project, and the syntax is simple and easy to use and maintain. These and other features will be discussed in detail in the following sections. A complete example of the use of MPC is shown in section 4.3.3.8.

4.2 Using MPC

An MPC project is a set of parameters that describe an individual build target (such as a library or executable). These parameters include the target name, include paths, source files, header files, etc. One or more projects can be defined within a single mpc file. An MPC workspace is just an arbitrary collection of projects.

Projects can be generated (without workspaces) by using the mpc.pl script. Multiple mpc files can be passed to this script. If no mpc files are passed to the script, it will search for project-related files (such as source files, header files, etc.) and incorporate them into a default project.
Figure 4-1 shows a high-level view of project file generation using mpc.pl.

To generate workspaces, you must run mwc.pl. This script will generate projects from mpc files and create a workspace based on those mpc files. If no mwc files are passed to the script, it will search in the current directory and its subdirectories for all mpc files and incorporate them into a single workspace.

For make based project types (make, gnuace, bmake, nmake), a workspace is just a top-level makefile. But, for graphical interfaces such as Visual Studio, a workspace is the top-level file that groups all of the project files together.
Figure 4-2 shows a high-level view of workspace file generation using mwc.pl.

![Diagram](Figure 4-2 Generating workspaces with mwc.pl)

### 4.2.1 Supported Build Tools

MPC generates workspaces and projects for use with many build tools. Table 4-1 lists the MPC types (used with mpc’s `-type` option) and their associated build tools.

**Table 4-1 MPC Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Build Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>automake</td>
<td>GNU Automake.</td>
</tr>
<tr>
<td>bmake</td>
<td>Borland Make.</td>
</tr>
<tr>
<td>cbx</td>
<td>Support for Borland CBuilderX is incomplete.</td>
</tr>
<tr>
<td>em3</td>
<td>eMbedded Visual C++ 3.00 and 4.00.</td>
</tr>
<tr>
<td>ghs</td>
<td>Support for Green Hills C++ Builder is incomplete.</td>
</tr>
<tr>
<td>gnuace</td>
<td>GNU Make for ACE/TAO only (ACE/TAO extension).</td>
</tr>
<tr>
<td>make</td>
<td>Generic make. The makefiles generated by this project type can be used with any version of make. However, due to configuration issues, it should not be used with ACE or TAO.</td>
</tr>
<tr>
<td>nmake</td>
<td>Microsoft NMake.</td>
</tr>
<tr>
<td>sle</td>
<td>Support for Visual SlickEdit is incomplete.</td>
</tr>
<tr>
<td>vc6</td>
<td>Visual C++ 6.0.</td>
</tr>
<tr>
<td>vc7</td>
<td>Visual C++ 7.0.</td>
</tr>
<tr>
<td>vc71</td>
<td>Visual C++ 7.1.</td>
</tr>
<tr>
<td>vc8</td>
<td>Visual C++ 8.0.</td>
</tr>
</tbody>
</table>
4.2.2 Command Line

The command line options for the workspace creator (mwc.pl) and the project creator (mpc.pl) are exactly the same. The project creator is used to generate one or more separate projects by passing mpc files to it on the command line. The workspace creator is used to generate one or more workspaces and the projects related to those workspaces.

Table 4-2 describes each option with the more commonly used options in bold and project specific options in italics.

Table 4-2 Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-base</td>
<td>This option allows the user to force any project to inherit from a specified base project. This option can be used multiple times to force multiple inheritance upon a project.</td>
</tr>
<tr>
<td>-exclude</td>
<td>If this option is used with mwc.pl, the directories or mwc files provided in a comma separated list will be excluded when searching for mpc files. Each element provided for exclusion should be relative to the starting directory. This option has no effect when used with mpc.pl.</td>
</tr>
<tr>
<td>-expand_vars</td>
<td>This option instructs MPC to perform direct replacement of $() variables with the values from the environment (if the -use_env option is used) or the values specified by the -relative option.</td>
</tr>
<tr>
<td>-feature_file</td>
<td>This option allows the user to override the default feature file (MPC/config/default.features or ACE_wrappers/bin/MakeProjectCreator/config/default.features) which may or may not exist. This file can be used to override feature values specified in the global.features file located in the config directory. Feature files are described in section 4.3.2.3.</td>
</tr>
<tr>
<td>-features</td>
<td>Specifies the feature list to set before processing. This is a comma separated list and should contain no spaces.</td>
</tr>
<tr>
<td>-genins</td>
<td>This option instructs MPC to generate an “install” file after processing each project. These “install” files can be used with the prj_install.pl script which will copy portions of the project related files into a user specified location.</td>
</tr>
<tr>
<td>-global</td>
<td>This option specifies the global input file. Values stored within this base project are applied to all generated projects. The default value is ACE_wrappers/bin/MakeProjectCreator/global.mpb or MPC/config/global.mpb.</td>
</tr>
</tbody>
</table>
Table 4-2 Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-hierarchy</td>
<td>If this option is used with mwc.pl, it will generate a workspace at each directory between the directory in which it is run and the location of a processed mpc file. This option has no effect when used with mpc.pl and is the default for “make” based workspace types.</td>
</tr>
<tr>
<td>-include</td>
<td>Include search directories are added with this option. These search directories are used when locating base projects, template input files and templates. It can be used multiple times on the same command line.</td>
</tr>
<tr>
<td>-language</td>
<td>This option is used to specify which language to assume when generating projects. The default language is cplusplus, but csharp, java and vb are also supported.</td>
</tr>
<tr>
<td>-make_coexistence</td>
<td>Make based project types that normally name the workspace Makefile (bmake or nmake) will name the generated output files such that they can coexist within the same directory. In essence, the bmake and nmake workspace names will not be Makefile, but the name of the workspace followed by the project type (.bmake or .nmake).</td>
</tr>
<tr>
<td>-name_modifier</td>
<td>This option allows the user to modify the output names of projects and workspaces. These are usually determined by either the mpc or mwc file, but can be modified using a pattern replacement. The parameter passed to this option will be used as the pattern and any asterisks (*) found in the pattern will be replaced with the project or workspace name depending on which type of file is being created.</td>
</tr>
<tr>
<td>-apply_project</td>
<td>This option is only useful with the -name_modifier option. When used in conjunction with -name_modifier, the pattern will be applied to the project name in addition to the project or workspace name.</td>
</tr>
<tr>
<td>-norefdefs</td>
<td>This option specifies that the default relative definitions should not be generated. See the -relative option below.</td>
</tr>
<tr>
<td>-notoplevel</td>
<td>This option tells mwc.pl to generate all workspace related project files, but do not generate the associated workspace. This option tells mpc.pl to process all mpc files passed in, but it will not generate any project files.</td>
</tr>
<tr>
<td>-recurse</td>
<td>Search from the current directory for any input files and process them from the directory in which they are located.</td>
</tr>
</tbody>
</table>
4.2 Using MPC

4.2.2.1 Additional Option Descriptions

Some of the options in Table 4-2 require an expanded explanation. You will find more information on the `-relative`, `-ti`, `-value_project` and `-value_template` options below.

Table 4-2 Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-relative</code></td>
<td>Relative paths are used to replace variables enclosed with $(). By default, any environment variable that ends in <code>_ROOT</code> will be automatically used as a relative path replacement. For more information see “The -relative Option.” on page 40.</td>
</tr>
<tr>
<td><code>-template</code></td>
<td>This option allows a user to specify an alternate template. Each project type has its own template and this option allows a user to override the default template.</td>
</tr>
<tr>
<td><code>-ti</code></td>
<td>Each project type has a set of template input files. With this option the default template input file can be overridden for a particular project type. For more information see “The -ti Option.” on page 40.</td>
</tr>
<tr>
<td><code>-type</code></td>
<td>This option specifies the type of project or workspace to be generated. It can be specified multiple times to generate different project types for a single set of input files.</td>
</tr>
<tr>
<td><code>-use_env</code></td>
<td>This option instructs MPC to replace all $() instances with the corresponding environment variable value instead of using values provided by the -relative option.</td>
</tr>
<tr>
<td><code>-static</code></td>
<td>Specifies that static project files will be generated from the MPC projects. The default is to generate dynamic project files.</td>
</tr>
<tr>
<td><code>-value_project</code></td>
<td>Use this option to override an mpc project assignment from the command line. This can be used to introduce new name value pairs to a project. However, it must be a valid project assignment. For more information see “The -value_project Option.” on page 41.</td>
</tr>
<tr>
<td><code>-value_template</code></td>
<td>This option can be used to override existing template input variable values from the command line. It can not be used to introduce new template input name value pairs. For more information see “The -value_template Option.” on page 41.</td>
</tr>
<tr>
<td><code>-version</code></td>
<td>The MPC version is printed and no files are processed.</td>
</tr>
<tr>
<td><code>-complete</code></td>
<td>The previously undocumented complete option can be used to generate a tcshe complete command that allows a user of the tcshe shell to complete on options as well as file names.</td>
</tr>
</tbody>
</table>
The -relative Option.
Some project types do not (completely) support the idea of accessing environment variables through the use of $(), and therefore MPC must ensure that generated projects are usable in these cases. In order to avoid the existence of $() variables within the generated project files, relative paths are put in place of those (where possible).

The -relative option takes a single parameter of a name value pair, for example:

```
mwc.pl -relative PROJ_TOP=/usr/projects/top
```

In above example, if the text "$(PROJ_TOP)" is found as a value for any mpb, mpc, mpd, or mpt variable then it is replaced by a path that is relative to /usr/projects/top. For example, if an mpc file located under /usr/projects/top/dir contained the following:

```
project {
    includes += $(PROJ_TOP)
}
```

The generated project file would contain text similar to:

```
CPPFLAGS += -I..
```

The $(PROJ_TOP) string was replaced with a directory value that is relative to the directory in which the mpc file is located.

The -ti Option.
The -ti option allows you to identify different template input files based on the type of target being built. Template input files correspond to four different categories: dll, lib, dll_exe, and lib_exe. Not all project types distinguish between the different categories, but the templates for various project types will be combined with different template input files, depending on the build target type, to generate different output.

To override the default template input file names, a -ti option is provided. The -ti option takes a single parameter of the form type:file. The type is one of the four categories stated above and the file is the base name of an mpt file located somewhere in the include search paths.
The following example shows a usage of the -ti option. It says that when generating a static project (lib), use the vc7lib template input file and when generating a dynamic project (dll), use the vc7dll template input file:

```plaintext
mpc.pl -type vc71 -ti lib:vc7dsplib -ti dll:vc7dspdll
```

These happen to be the default values for the vc71 type, but it illustrates that a different template input can be specified for each category.

**The -value_project Option.**
The -value_project option can be useful when the need arises to modify the value of an MPC variable across one or more mpc files. For example, if you wanted to generate all of your projects with an additional include search path you would run the following command:

```plaintext
mwc.pl -value_project includes+=/include/path
```

In the above example, an additional include search path of /include/path would be placed in all generated projects.

**The -value_template Option.**
This option modifies existing or adds new template input name/value pairs. For example, if you wanted to generate dynamic vc71 projects with only Release targets, you would run the following command:

```plaintext
mwc.pl -type vc71 -value_template configurations=Release
```

To find out what template input variables are defined, see the individual mpd file of interest ($ACE_ROOT/bin/MakeProjectCreator/templates/*.mpd and $MPC_ROOT/templates/*.mpd) and search for names used within <% and %>. Names that are not listed as project keywords (Table 4-3 on page 46) are template variables.

### 4.2.3 Environment Variables
MPC recognizes a few environment variables that alter the way it performs certain tasks. The sections below describe each one and the effect it has on MPC.
MPC will use the options defined in \texttt{MPC\_COMMANDLINE} as if they were given on the command line to mwc.pl or mpc.pl. The environment value will be prepended to options actually passed to mwc.pl or mpc.pl on the actual command line.

The \texttt{MPC\_DEPENDENCY\_COMBINED\_STATIC\_LIBRARY} environment variable only affects the way workspace dependencies are created for static projects with the \texttt{em3}, \texttt{vc6}, \texttt{vc7}, \texttt{vc71} and \texttt{vc8} project types. If this environment variable is set, MPC will generate inter-project dependencies for libraries within a single workspace. This is usually not desired since adding these dependencies in a static workspace has the side effect of including dependee libraries into the dependent library.

If the \texttt{MPC\_LOGGING} environment variable is set, MPC will parse the value and provide informational, warning and diagnostic messages depending on it’s setting. If the value contains \texttt{info=1}, informational messages will be printed. If it contains \texttt{warn=1}, warning messages will be printed. If it contains \texttt{diag=1}, diagnostic messages will be printed. And lastly, if it contains \texttt{detail=1}, detail messages will be printed. If it contains none of these, MPC will act as if \texttt{MPC\_SILENT} was set.

The \texttt{MPC\_SILENT} environment variable instructs MPC not to print any messages, except error messages. The progress indicator is still printed.

If \texttt{MPC\_VERBOSE\_ORDERING} is set, MPC will warn the user about references to projects in the “after” keyword that have not been processed. This only has an effect when running mwc.pl.

### 4.3 Writing MPC and MWC Files

You may want to familiarize yourself with the various input files for MPC. The input file types and the syntax of each are discussed in the sections below.

#### 4.3.1 Input Files

There are four different input files associated with MPC. For most users of MPC, the main files of concern are mpc and mwc files.
4.3 Writing MPC and MWC Files

4.3.1.1 Project Files (mpc)
Project files, those with the .mpc extension, contain such things as include paths, library paths, source files and inter-project dependencies. An mpc file can contain one or more “projects” each of which needs to be uniquely named to avoid project generation errors. Projects represent build targets such as libraries and executables.

4.3.1.2 Workspace Files (mwc)
Workspaces are defined by providing a list of mpc files, directories or other mwc files in a single mwc file. For each mpc file, the Workspace Creator calls upon the Project Creator to generate the project. After all of the projects are successfully generated, the tool-specific workspace is generated containing the projects and any defined inter-project dependency information (if supported by the build tool). An mwc file can contain one or more “workspaces,” each of which needs to be uniquely named. If no workspace files are provided to the workspace creator, the current directory is traversed and any mpc files located will be part of the workspace that is generated.

4.3.1.3 Base Project Files (mpb)
One of the many unique and useful features of MPC is that the project definition files can use inheritance. Project inheritance allows a user to set up a base project (mpb file) that can contain information that is applicable to all derived projects. Common project attributes, such as include paths, library paths, and inter-project dependencies, could be described in this base project and any project that inherits from it would contain this information as well.

4.3.1.4 Base Workspace Files (mwb)
As with projects, workspaces can also inherit from other workspaces. A base workspace can provide workspace information that may be common to other workspaces.

4.3.2 General Input File Syntax
In this section we discuss the syntax of the various files. We also describe some of the default values that go along with these files.
4.3.2.1 mwc and mwb

Workspaces can contain individual mpc files or directories. There can be one or more workspaces defined within a single mwc file.

```
workspace(optional name): optional_base_workspace {
    file.mpc
    directory
    other.mwc

    exclude(vc6, vc7, vc71, vc8, nmake) {
        this_directory
    }
}
```

A workspace can be given a name. This is the value given in the parentheses after the keyword `workspace`. If the workspace is not given a name, the workspace name is taken from the name of the mwc file without the extension.

Workspaces can also inherit from other workspaces. In the above example, `optional_base_workspace` would be the base name of an mwb file with no extension that contains workspace information. This information would then be included in each workspace that inherits from it.

The lines between the curly braces contain assignments, mpc files, directories, other workspace files or exclusion sections. The mpc files listed will be included in the workspace. If a directory is listed within the workspace, the workspace creator will recursively traverse that directory and use any mpc files that are found. If a workspace file is listed it will be aggregated into the main workspace.

A workspace can have assignments interspersed within the directories and mpc files. These assignments modify the way projects are generated.

The `cmdline` setting can be used to provide command line options that would normally be passed to `mwc.pl` (see Table 4-2). However, the `-type`, `-recurse`, `-noreldefs`, `-make_coexistence`, `-genins`, `-into` and `-language` options as well as input files are ignored. Environment variables may be accessed through `${NAME}`, where `NAME` is the environment variable name. The `cmdline` assignment may be useful for workspaces that require specific `mwc.pl` options in order to process correctly.

The only other setting supported by `mwc.pl` is `implicit`. If `implicit` is set to 1 then default project files are generated in each directory where no mpc file exists. The `implicit` keyword can also be set to the name of a base
project. In this case, the implicitly generated project will inherit from the base project specified in the assignment. Either way, if the directory does not contain files that can be used within a project, no project is created. Setting implicit can be useful when you want to define specific workspaces, but the MPC defaults are sufficient for the directories involved within the workspace.

Scoped assignments are assignments that are associated with specific mpc files or directories listed with the scope of the assignment. The following example shows a scoped assignment of cmdline that only applies to one of the mpc files listed in the workspace. In this example, directory/foo.mpc would be processed as if the -static option had been passed on the command line whereas other directories and mpc files would not.

workspace {
  ...
  static {
    cmdline += -static
directory/foo.mpc
  }
  exclude(gnuace, make) {
    some.mpc
  }
}

Exclusion sections are used to prevent directories and mpc files from being processed. These excluded directories and mpc files will be skipped when generating project files and workspaces. The exclude keyword accepts project types within the parentheses (as above), which will cause the workspace creator to only exclude the listing for particular types. If no types are provided, exclusion will take place for all project types.

Comments are similar to the C++ style comments. Any text after a double slash (//) is considered a comment.

### 4.3.2.2 mpc and mpb

**Project Declarations**

Project declarations are similar to workspace declarations, but are a bit more complex. An mpc file can contain one or more “projects” and each project can inherit from base projects.
The Makefile, Project, and Workspace Creator (MPC)

```plaintext
project(optional name): base_project, another_base_project {
  exename = client
  includes += directory_name other_directory
  libpaths += /usr/X11R6/lib

  Header_Files {
    file1.h
    file2.h
    fileN.h
  }

  Source_Files {
    file1.cpp
    file2.cpp
    fileN.cpp
  }
}
```

If the optional project name is not given, then the project name is taken from the name of the mpc file without the extension. Therefore, if your mpc file is going to contain multiple projects, it is important to provide project names to prevent each generated project from overwriting the other. MPC will issue an error and stop if duplicate project names are detected.

**Base Projects**

Base projects can be of the extension `.mpb` and `.mpc`. If a file with the name of the base project with an `.mpb` or `.mpc` extension cannot be found within the mpc include search path, a fatal error is issued and processing halts.

**Assignment Keywords**

Table 4-3 shows the keywords that can be used in an assignment (i.e. `=`, `+=` or `-=`) within an mpc file. The most commonly used keywords are shown in bold face.

**Table 4-3 Assignment Keywords**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>after</em></td>
<td>Specifies that this project must be built after 1 or more project names listed.</td>
</tr>
<tr>
<td><em>avoids</em></td>
<td>Specifies which features should be disabled in order to generate the project file. Under the GNUACE type, it also specifies which make macros should not be set to build the target.</td>
</tr>
</tbody>
</table>
4.3 Writing MPC and MWC Files

Table 4-3 Assignment Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>custom_only</td>
<td>This setting instructs MPC to create projects that only contain custom generation targets. Any files included in the projects will be provided by custom component lists defined through the use of Define_Cust.</td>
</tr>
<tr>
<td>dynamicflags</td>
<td>Specifies preprocessor flags passed to the compiler when building a dynamic library.</td>
</tr>
<tr>
<td>dllout</td>
<td>If defined, specifies where the dynamic libraries will be placed. This overrides libout in the dynamic case.</td>
</tr>
<tr>
<td>exename</td>
<td>Determines that the project will be an executable and the name of the executable target.</td>
</tr>
<tr>
<td>includes</td>
<td>Specifies one or more directories to supply to the compiler for use as include search paths.</td>
</tr>
<tr>
<td>install</td>
<td>Specifies where executables will be installed.</td>
</tr>
<tr>
<td>libout</td>
<td>Specifies where the dynamic and static libraries will be placed.</td>
</tr>
<tr>
<td>libpaths</td>
<td>Specifies one or more directories to supply to the compiler for use as library search paths.</td>
</tr>
<tr>
<td>libs</td>
<td>Specifies one or more libraries to link into the target. Library modifiers may be added when being processed in the template file. For example, library modifiers are added when using the vc6 project type.</td>
</tr>
<tr>
<td>lit_libs</td>
<td>This is the same as libs except that a library modifier will not be added.</td>
</tr>
<tr>
<td>macros</td>
<td>Values supplied here will be passed directly to the compiler as command line defined macros.</td>
</tr>
<tr>
<td>pch_header</td>
<td>The name of the precompiled header file. See the discussion below this table for more information.</td>
</tr>
<tr>
<td>pch_source</td>
<td>The name of the precompiled source file. See the discussion below this table for more information.</td>
</tr>
<tr>
<td>pure_libs</td>
<td>This is similar to litlibs except that no prefix or extension is added to the names specified.</td>
</tr>
<tr>
<td>postbuild</td>
<td>If this is defined in the project, the value will be interpreted as commands to run after the project has been successfully built. The <code>&lt;% %&gt;</code> construct (See “Template Files (mpd)” on page 66.) can be used within this value to access template variables and functions of the template parser.</td>
</tr>
</tbody>
</table>
Assignments can also use the `+=` and `-=` operators to add and subtract values from keyword values.

If a `sharedname` is specified in the `mpc` file and `staticname` is not used, then `staticname` is assumed to be the same as `sharedname`. This also applies in the opposite direction.

If neither `exename`, `sharedname` nor `staticname` is specified, MPC will search the source files for a `main` function. If a `main` is found, the `exename` will be set to the name of the file, minus the extension, that contained the `main` function. Otherwise, `sharedname` and `staticname` will be set to the project name.

If the project name, `exename`, `sharedname` or `staticname` contain an asterisk it instructs MPC to dynamically determine a portion of the name based on certain defaults. If the project name contains an asterisk, then the asterisk will be replaced with the default project name. If `exename`, `sharedname` or

### Table 4-3 Assignment Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recuse</code></td>
<td>If set to 1, MPC will recurse into directories listed under component listings (such as <code>Source_Files</code>, <code>Header_Files</code>, etc.) and add any component corresponding files to the list. This keyword can be used as a global project setting or a component scoped setting.</td>
</tr>
<tr>
<td><code>requires</code></td>
<td>Specifies which features should be enabled in order to generate the project file. Under the GNUACE type, it also specifies which make macros should be set to build the target.</td>
</tr>
<tr>
<td><code>sharedname</code></td>
<td>Determines that the project will be a library and the name of the dynamic library target. See the discussion below this table for more information.</td>
</tr>
<tr>
<td><code>staticflags</code></td>
<td>Specifies preprocessor flags passed to the compiler when building a static library.</td>
</tr>
<tr>
<td><code>staticname</code></td>
<td>Determines that the project will be a library and the name of the static library target.</td>
</tr>
<tr>
<td><code>tagchecks</code></td>
<td>For GNUACE Make only, specifies one or more names to search for in the macros specified by <code>tagname</code>.</td>
</tr>
<tr>
<td><code>tagname</code></td>
<td>Specifies the GNUACE Make macro to check before building the target.</td>
</tr>
<tr>
<td><code>version</code></td>
<td>Specifies the version number for the library or executable.</td>
</tr>
</tbody>
</table>
4.3 Writing MPC and MWC Files

static name contains an asterisk, then the asterisk will be replaced with the project name.

If the pch_header keyword is not used and a file exists, in the directory in which the mpc file is located, that matches *_pch.h it is assumed to be the precompiled header for that directory. If there are multiple pch files in the directory, then the precompiled header that closely matches the project name will be chosen. Similar logic applies for the pch_source keyword.

Components

An mpc file can also specify the files to be included in the generated “project” file. These files are specified using the component names shown in Table 4-4. However, most of the time users will want to allow MPC to provide the default values for project files.

Table 4-4 Component Names and Default Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source_Files</td>
<td>Defaults to all files in the directory that have the following extensions: cpp, cxx, cc, c, and C.</td>
</tr>
<tr>
<td>Header_Files</td>
<td>Defaults to all files in the directory that have the following extensions: h, hpp, hxx, and hh.</td>
</tr>
<tr>
<td>Inline_Files</td>
<td>Defaults to all files in the directory that have the following extensions: i and inl.</td>
</tr>
<tr>
<td>Template_Files</td>
<td>Defaults to all files in the directory that end in the following: _T.cpp, _T.cxx, _T.cc, _T.c, and _T.C.</td>
</tr>
<tr>
<td>Documentation_Files</td>
<td>Defaults to all files in the directory that match the following: README, readme, .doc, .html and .txt.</td>
</tr>
<tr>
<td>Resource_Files</td>
<td>Defaults to all files in the directory that match the project name and have an rc extension.</td>
</tr>
</tbody>
</table>

If a component is not specified in the mpc file, the default value will be used. To disallow a particular set of files that may exist in the directory, you must declare an empty set of the particular component type.

Each component name accepts two forms. The first form is a simple list of files within the construct.

Source_Files {
  file1.cpp
  file2.cpp
}
The second form is a complex list of files within named blocks.

Source_Files(MACRO_NAME) {
    BlockA {
        file1.cpp
        file2.cpp
    }
    BlockB {
        file3.cpp
        file4.cpp
    }
}

The second form allows the user to logically group the files to make future maintenance easier. Using this form has the effect of visually grouping files in the generated project file for the em3, gnuace, vc6, vc7, vc71 and vc8 project types.

If a file is listed in the Source_Files component list and a corresponding header or inline file exists in the directory, it is added to the corresponding component list unless it is already listed.

**Verbatim Clause**

The verbatim construct can be used to place text into a generated project file verbatim. The verbatim syntax is as follows:

verbatim(<project type>, <location>) {
    ..
}

When MPC is generating a project of type <project type> and encounters a marker in the template file (see Table 4-8 on page 67) that matches the <location> name, it will place the text found inside the construct directly into the generated project. If the text inside the construct requires that white space be preserved, each line must be enclosed in double quotes. The following verbatim example would result in gnuace generated projects having a rule at the bottom of the GNUmakefile where the all: target depends on foo.

verbatim(gnuace, bottom) {
    all: foo
}
4.3 Writing MPC and MWC Files

Specific Clause
The `specific` keyword can be used to define assignments that are specific to a particular project type. This will allow platform or OS-specific values to be placed into a project. For example, on one platform you may want to link in a library named `qt-mt`, but on another you need to link in `qt-mt230nc`.

```cpp
specific(bmake, nmake, vc6, vc7, vc71, vc8) {
    lit_libs += qt-mt230nc
} else {
    lit_libs += qt-mt
}
```

If an else clause is provided, it is required to be on the same line as the closing curly brace. You may also negate the project type (using '!') which will cause the specific to be evaluated for all types except the type specified.

If a keyword used within a `specific` section is not recognized as a valid MPC keyword, it is interpreted to be template value modifier. In this situation, this construct works exactly the same way as the `-value_template` command line option (see Table 4-2 on page 37).

Conditional Clause
This scope allows addition of source files conditionally based on a particular project type. The syntax is as follows:

```cpp
conditional(<project type> [, <project type> ...]) {
    source1.cpp
    ...
}
conditional(<project type> [, <project type> ...]) {
    source1.cpp
    ...
} else {
    source2.cpp
    ...
}
```

If the else is provided, it is required to be on the same line as the closing curly brace. You may also negate the project type (using '!'') which will cause the conditional to be evaluated for all types except the type specified.
Custom Types and Build Rules

MPC allows you to define your own custom file types to support a variety of custom build rules. Below is an example of a custom definition.

```
project {
    Define_Custom(MOC) {
        automatic = 0
        command = $(QTDIR)/bin/moc
        output_option = -o
        inputext = .h
        pre_extension = _moc
        source_outputext = .cpp
        keyword mocflags = commandflags
    }

    // Custom Component
    MOC_Files {
        QtReactor.h
    }

    Source_Files {
        QtReactor_moc.cpp
    }
}
```

The above example defines a custom file type, "MOC", that describes basic information about how to process the input files and what output files are created. Once the custom file type is defined, MOC_Files can be used to specify the input files for this new file type.

Table 4-5 contains the keywords that can be used within the scope of Define_Custom.

**Table 4-5 Define_Custom Keywords**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>If set to 1, then attempt to automatically determine which files belong to the set of input files for the custom type. If set to 0, then no files are automatically added to the input files. If omitted, then automatic is assumed to be 1. Custom file types that are automatic will have the side effect of possibly adding files to Source_Files, Inline_Files, Header_Files, Template_Files, Resource_Files and Documentation_Files depending on which extension types the command generates.</td>
</tr>
<tr>
<td>command</td>
<td>The name of the command that should be used to process the input files for the custom type.</td>
</tr>
</tbody>
</table>
### Table 4-5 Define_Custom Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commandflags</td>
<td>Any options that should be passed to the command.</td>
</tr>
<tr>
<td>dependent</td>
<td>If this is given a value, then a dependency upon that value will be placed upon all of the generated files. The default for this is unset and no dependency will be generated.</td>
</tr>
<tr>
<td>inputtext</td>
<td>This is a comma separated list of input file extensions that belong to the command.</td>
</tr>
<tr>
<td>keyword &lt;name&gt;</td>
<td>This is a special assignment that allows the user to map &lt;name&gt; into the project level namespace. The value (if any) that is assigned to this construct must be one of the keywords that can be used within a Define_Custom clause. The result of this assignment is the ability modify the value of keywords that are normally only accessible within the scope of a custom component (e.g. command, commandflags, etc.).</td>
</tr>
<tr>
<td>libpath</td>
<td>If the command requires a library that is not in the normal library search path, this keyword can be used to ensure that the command is able to find the library that it needs to run.</td>
</tr>
<tr>
<td>output_option</td>
<td>If the command takes an option to specify a single file output name, then set it here. Otherwise, this should be omitted.</td>
</tr>
<tr>
<td>pch_postrule</td>
<td>If this is set to 1, then a rule will be added to the custom rule that will modify the source output files to include the precompiled header file.</td>
</tr>
<tr>
<td>postcommand</td>
<td>This allows users to create arbitrary commands that will be run after the main command is run to process the custom input files.</td>
</tr>
<tr>
<td>pre_extension</td>
<td>If the command produces multiple files of the same extension, this comma separated list can be used to specify them. For example, tao_idl creates two types of files per extension (C.h, S.h, C.cpp, S.cpp, etc.) This applies to all extension types.</td>
</tr>
<tr>
<td>source_pre_extension</td>
<td>This is the same as pre_extension except that it only applies to source_outputext.</td>
</tr>
<tr>
<td>inline_pre_extension</td>
<td>This is the same as pre_extension except that it only applies to inline_outputext.</td>
</tr>
<tr>
<td>header_pre_extension</td>
<td>This is the same as pre_extension except that it only applies to header_outputext.</td>
</tr>
<tr>
<td>template_pre_extension</td>
<td>This is the same as pre_extension except that it only applies to template_outputtext.</td>
</tr>
</tbody>
</table>
Table 4-5 Define Custom Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource_pre_extension</td>
<td>This is the same as pre_extension except that it only applies to resource_outputext.</td>
</tr>
<tr>
<td>documentation_pre_extension</td>
<td>This is the same as pre_extension except that it only applies to documentation_outputext.</td>
</tr>
<tr>
<td>pre_filename</td>
<td>The syntax for this is the same as pre_extension, but the values specified are prepended to the file name instead of the extension. This applies to all extension types.</td>
</tr>
<tr>
<td>source_pre_filename</td>
<td>This is the same as pre_filename except that it only applies to source_outputext.</td>
</tr>
<tr>
<td>inline_pre_filename</td>
<td>This is the same as pre_filename except that it only applies to inline_outputext.</td>
</tr>
<tr>
<td>header_pre_filename</td>
<td>This is the same as pre_filename except that it only applies to header_outputext.</td>
</tr>
<tr>
<td>template_pre_filename</td>
<td>This is the same as pre_filename except that it only applies to template_outputext.</td>
</tr>
<tr>
<td>resource_pre_filename</td>
<td>This is the same as pre_filename except that it only applies to resource_outputext.</td>
</tr>
<tr>
<td>documentation_pre_filename</td>
<td>This is the same as pre_filename except that it only applies to documentation_outputext.</td>
</tr>
<tr>
<td>source_outputext</td>
<td>This is a comma separated list of possible source file output extensions. If the command does not produce source files, then this can be omitted.</td>
</tr>
<tr>
<td>inline_outputext</td>
<td>This is a comma separated list of possible inline file output extensions. If the command does not produce inline files, then this can be omitted.</td>
</tr>
<tr>
<td>header_outputext</td>
<td>This is a comma separated list of possible header file output extensions. If the command does not produce header files, then this can be omitted.</td>
</tr>
<tr>
<td>template_outputext</td>
<td>This is a comma separated list of possible template file output extensions. If the command does not produce template files, then this can be omitted.</td>
</tr>
<tr>
<td>resource_outputext</td>
<td>This is a comma separated list of possible resource file output extensions. If the command does not produce resource files, then this can be omitted.</td>
</tr>
<tr>
<td>documentation_outputext</td>
<td>This is a comma separated list of possible documentation file output extensions. If the command does not produce documentation files, then this can be omitted.</td>
</tr>
</tbody>
</table>
There is a special interaction between custom components and the source, header and inline components. If a custom definition is set to be “automatic” and custom component files are present but not specified, the default custom generated names are added to the source, header and inline component lists unless those names are already listed (or partially listed) in those component lists. See “Custom Types and Build Rules” on page 52 for more information about defining your own custom type.

Particular output extensions for custom build types are not required. However, at least one output extension type is required for MPC to generate a target. Your command does not necessarily have to generate output, but an extension type is required if you want the input file to be processed during the project compilation.

If the custom output can not be represented with the above output extension keywords (*_outputext) and you have knowledge of the output files a priori, you can represent them with the ’>>’ construct.

Below is an example that demonstrates the use of ’>>’. The command takes an input file name of foo.prp and produces two files that have completely unrelated filenames, hello.h and hello.cpp.

```
project {
  Define_Custom(Quogen) {
    automatic           = 0
    command             = perl quogen.pl
    commandflags        = --debuglevel=1 --language=c++ ...
    inputext            = .prp
    keyword quogenflags = commandflags
  }

  Quogen_Files {
    foo.prp >> hello.h hello.cpp
  }

  Source_Files {
    hello.cpp
  }
}
```

### Table 4-5 Define Custom Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic_outputext</td>
<td>If the command does not generate any of the other output types listed above, then the extensions should be listed under this.</td>
</tr>
</tbody>
</table>
You can use the ’<<’ construct to represent dependencies for specific custom input file. For instance, in the above example, assume that foo.prp depends upon foo.in, we would represent this by adding << foo.in as shown below.

```
Quogen_Files {
  foo.prp >> hello.h hello.cpp << foo.in
}
```

An additional construct can be used within the scope of a Define_Custom. This construct is called optional, and can be used to represent optional custom command output that is dependent upon particular command line parameters passed to the custom command.

```
project {
  Define_Custom(TEST) {
    optional(keyword) {
      flag_keyword(option) += value [, value]
    }
  }
}
```

In the above fragment, keyword can be any of the pre_extension, pre_filename keywords or any of the keywords that end in _outputext. The flag_keyword can be any of the custom definition keywords, however only commandflags has any functional value. The flag_keyword value is searched for the option value contained inside the parenthesis. If it is found the value or values after the += are added to the list specified by keyword. This can also be negated by prefixing the option with an exclamation point (!).

The example below shows how the optional construct is used by the custom definition for the tao_idl command (see ACE_wrappers/bin/MakeProjectCreator/config/taoidldefaults.mp). The -GA option causes tao_idl to generate an additional source file (based on the idl file name) with an A.cpp extension. The -Sc option causes tao_idl to suppress the generation of S_T related files.

```
Define_Custom(IDL) {
  ...
  inputext = .idl
  source_pre_extension = C, S
  header_pre_extension = C, S
  inline_pre_extension = C, S
  source_outputext = .cpp, .cxx, .cc, .C
  header_outputext = .h, .hpp, .hxx, .hh
}```
4.3 Writing MPC and MWC Files

inline_outputext = .inl, .i
keyword idlflags = commandflags

optional(source_pre_extension) {
    commandflags(-GA) += A
}
optional(template_outputext) {
    commandflags(!-Sc) += S_T.cpp, S_T.cxx, S_T.cc, S_T.C
}
optional(header_pre_extension) {
    commandflags(!-Sc) += S_T
}
optional(inline_pre_extension) {
    commandflags(!-Sc) += S_T
}

For custom file types, there are a few keywords that can be used within the custom file type component lists: command, commandflags, dependent, gendir, postcommand, and recurse.

The recurse keyword works as described in Table 4-3, “Assignment Keywords”.

The command, commandflags, dependent and postcommand keywords can be used to augment or override the value defined in the Define_Custom section.

The gendir keyword can be used (only if output_option is set in Define_Custom) to specify the directory in which the generated output will go. Here is an example:

MOC_Files {
    commandflags += -nw
    gendir = moc_generated
    QtReactor.h
}

Source_Files {
    moc_generated/QtReactor_moc.cpp
}

In the above example, the -nw option is added to commandflags and the generated file (QtReactor_moc.cpp) is placed in the moc_generated directory. If the MOC custom definition did not have an output_option setting, then options would need to be added to commandflags or a
postcommand would need to be defined to ensure that the output actually went into the moc_generated directory.

**Custom Post Command**

When defining a postcommand as part of a Define_Custom, a few pseudo template variables are available to provide some flexibility. The following table shows the pseudo template variables that can be accessed only from the postcommand. Please note that < and > are part of the syntax.

**Table 4-6 Post Command Pseudo Variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%=input%&gt;</td>
<td>The input file for the original command.</td>
</tr>
<tr>
<td>&lt;%=input_basename%&gt;</td>
<td>The basename of the input file for the original command.</td>
</tr>
<tr>
<td>&lt;%=input_noext%&gt;</td>
<td>The input file for the original command with the extension stripped off.</td>
</tr>
<tr>
<td>&lt;%=input_ext%&gt;</td>
<td>This gives the file extension of the input file (if there is one).</td>
</tr>
<tr>
<td>&lt;%=output%&gt;</td>
<td>The output file created by the original command.</td>
</tr>
<tr>
<td>&lt;%=output_basename%&gt;</td>
<td>The basename of the output file for the original command.</td>
</tr>
<tr>
<td>&lt;%=output_noext%&gt;</td>
<td>The output file created by the original command with the extension stripped off.</td>
</tr>
<tr>
<td>&lt;%=output_ext%&gt;</td>
<td>This gives the file extension of the output file (if there is one).</td>
</tr>
</tbody>
</table>

The output file can be referenced as a generic output file, or it can be referenced as a component file using one of the following variables. If it does not match the particular type the value will be empty.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%=source_file%&gt;</td>
<td>The output file if it has a source file extension.</td>
</tr>
<tr>
<td>&lt;%=template_file%&gt;</td>
<td>The output file if it is a template file.</td>
</tr>
<tr>
<td>&lt;%=header_file%&gt;</td>
<td>The output file if it has a header file extension.</td>
</tr>
<tr>
<td>&lt;%=inline_file%&gt;</td>
<td>The output file if it has an inline file extension.</td>
</tr>
<tr>
<td>&lt;%=documentation_file%&gt;</td>
<td>The output file if it is a documentation file.</td>
</tr>
<tr>
<td>&lt;%=resource_file%&gt;</td>
<td>The output file if it has a resource file extension.</td>
</tr>
</tbody>
</table>
The following table describes the pseudo template variables that can be used in the command, commandflags, dependent, output_option and postcommand settings.

Table 4-7 Common Pseudo Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%temporary%&gt;</td>
<td>A temporary file name. The generated temporary file name contains no directory portion and is the same for each use within the same variable setting.</td>
</tr>
<tr>
<td>&lt;%cat%&gt;</td>
<td>A platform non-specific command to print a file to the terminal.</td>
</tr>
<tr>
<td>&lt;%cp%&gt;</td>
<td>A platform non-specific command to copy a file.</td>
</tr>
<tr>
<td>&lt;%mkdir%&gt;</td>
<td>A platform non-specific command to make a directory.</td>
</tr>
<tr>
<td>&lt;%mv%&gt;</td>
<td>A platform non-specific command to move a file.</td>
</tr>
<tr>
<td>&lt;%rm%&gt;</td>
<td>A platform non-specific command to delete a file.</td>
</tr>
<tr>
<td>&lt;%null%&gt;</td>
<td>A platform non-specific null device.</td>
</tr>
<tr>
<td>&lt;%gt%&gt;</td>
<td>A platform and project non-specific representation of a greater than sign.</td>
</tr>
<tr>
<td>&lt;%lt%&gt;</td>
<td>A platform and project non-specific representation of a less than sign.</td>
</tr>
<tr>
<td>&lt;%quote%&gt;</td>
<td>A project non-specific representation of a double quote.</td>
</tr>
<tr>
<td>&lt;%and%&gt;</td>
<td>A platform and project non-specific representation of a command conditional and.</td>
</tr>
<tr>
<td>&lt;%or%&gt;</td>
<td>A platform and project non-specific representation of a command conditional or.</td>
</tr>
</tbody>
</table>

4.3.2.3 The Feature File

The term feature, as used by MPC, describes different concepts or external software that a project may require in order to build properly. The feature file determines which features are enabled or disabled which has a direct effect on whether or not MPC generates a project.

It supports the standard comment (//) and assignment of numbers to feature names. These feature names will correspond to values given to the requires and avoids keywords in mpc files.

If a feature is not listed in the feature file or is listed with a boolean value of true (1), that feature is enabled. If a feature is listed and has a boolean value of false (0), that feature is disabled.
If a feature name is listed in the requires value for a particular project and that feature is enabled, that project will be generated. If the feature is not enabled, the project will not be generated.

The opposite holds true for the avoids keyword. If a feature name is listed in the avoids value for a project and the feature is disabled, that project will be generated. If the feature is enabled, the project will not be generated.

The global feature file for MPC contains the following values.

```plaintext
boost = 0
mfc = 0
qt = 0
rpc = 0
zlib = 0
zzip = 0
```

In the above contents, boost, mfc, qt, rpc, zlib and zzip are disabled for each project generated. If these values do not suit your needs, then you must do one of three things:

- Create a project specific feature file in the config directory (ex. make.features) to set features for a particular project type.
- Create a default.features file in the config directory that contains the feature set you need.
- Create a feature file anywhere you like with the features you want and use the -feature_file option to specify the location.
- Use the -features option to dynamically modify the feature settings.

Generated projects will have a combination of features specified in the global.features file as well as in your feature file. Therefore, if a feature is disabled in the global file and you want to enable it, you must explicitly enable it in your feature file.

### 4.3.2.4 Feature Projects

A feature project contains information as a project would, but can only be a base project and will only be added to a sub project if the features that it requires are enabled or the features that it avoids are disabled.

A feature definition requires at least one feature name. A name by itself specifies that the feature must be enabled. A ’!’ in front of the feature name
indicates that the feature must be disabled. There may be more than one comma separated feature listed between the parenthesis.

The following example show how to declare a feature project.

```c
// ziparchive.mpb
feature(ziparchive) {
    includes += $(ZIPARCHIVEROOT)
    libpaths += $(ZIPARCHIVEROOT)/lib
    libs     += ziparch
}
```

With this example, any project that inherits from the `ziparchive` base feature project will contain the project information only if the `ziparchive` feature is enabled.

### 4.3.3 Defaults

MPC has been designed to minimize the amount of maintenance that goes into keeping build tool files up-to-date with the project. If your source code is organized properly, the maintenance of your mpc files should be minimal.

With the use of inheritance and proper code arrangement, an mpc file for a TAO related project may be as simple as:

```c
project: taoserver {
}
```

This project definition could be used to generate a project for a TAO server with multiple idl, header and source files.

The idea of proper source layout is basically summarized as one directory per binary target. If only the files that pertain to a single target are located in the directory with the mpc file, then the MPC defaults will satisfy most project needs.

Of course, it will not always be possible or desirable to organize your project code in this fashion, so all defaulting behavior can be overridden. The next sections describe the default behaviors of MPC and how to override them.

### 4.3.3.1 Source Files

New source files are added and others are removed quite often in a developing project. If the `Source Files` component is left out of an mpc file, then MPC will assume that any file matching one of the `source` extensions is to be included in the project. For most project types, the source extensions are:
.cpp, .cxx, .cc, .c and .C. Only the following extensions are considered source extensions: .cpp, .cxx and .c for the vc6 project type as Visual C++ 6.0 does not understand files with the .cc or .C extension.

### 4.3.3.2 Template Files

MPC assumes that any file matching one of the _template_ extensions is to be included in the project if the Template_Files component is left out of an mpc file. For most project types, the template extensions are: _T.cpp, _T.cxx, _T.cc, _T.c and _T.C. However, only the _T.cpp and _T.cxx extensions are considered template extensions for the vc6 project type.

If the Source_Files component is defaulted, and a file is explicitly listed in the Template_Files section that happens to appear to MPC as a source file (i.e. has a source file extension, but does not have _T directly before it), MPC will automatically exclude it from the Source_Files component.

### 4.3.3.3 Inline Files

As with source files, the Inline_Files component can be left out of an mpc file to allow it to generate defaults. Files that match the .i and .inl extensions are considered inline files.

The Inline_Files component has a special interaction with the Source_Files component. If the Source_Files component has files listed and the Inline_Files component is omitted, then each source file is _matched_ to an inline file. If the matching inline file is found or would be generated from a custom command, it is added to the Inline_Files component list.

### 4.3.3.4 Header Files

As with source files, the Header_Files component can be left out of an mpc file to allow it to generate defaults. Files that match the .h, .hpp, .hxx, and .hh extensions are considered header files.

The Header_Files component has a special interaction with the Source_Files component. If the Source_Files component has files listed and the Header_Files component is omitted, then each source file is _matched_ to a header file. If the matching header file is found or would be generated from a custom command, then it is added to the Header_Files component list.
4.3.3.5 Documentation Files

The Documentation Files component, if omitted, will default to all files that end in the following: README, readme, .doc, .html and .txt.

4.3.3.6 Resource Files

The Resource Files component, if omitted, will default to only the files that end in .rc and are similar to the name of the project. For example, if a directory contains three .rc files and the project name is foo, only the .rc files that contain the word foo will automatically be added to the Resource Files component list.

4.3.3.7 Custom Defined Files

The Custom Defined Files components have a special interaction with the Source Files component. If the custom command generates source files and has the automatic setting set to 1, they will automatically be added to the Source Files component list. If any of the files listed in the Source Files components list match any of the generated source file names, then none of the generated source file names will be automatically added to the Source Files components list.

4.3.3.8 Example MPC File

The example below uses the directory contents of $TAO_ROOT/orbsvcs/performance-tests/RTEvent/lib to illustrate the simplicity of mpc files:

Auto_Disconnect.cpp         Loopback_Supplier.h       RTEC_Initializer.cpp
Auto_Disconnect.h           Low_Priority_Setup.cpp     RTEC_Initializer.h
Auto_Disconnect.inl         Low_Priority_Setup.h      rtec_perf_export.h
Auto_Functor.cpp            Low_Priority_Setup.inl    RTEC_Perf.mpc
Auto_Functor.h              Makefile                  RTFOA_Setup.cpp
Auto_Functor.inl            ORB_Holder.cpp           RTFOA_Setup.h
Client_Group.cpp            ORB_Holder.h              RTFOA_Setup.inl
Client_Group.h              ORB_Holder.inl            RTE_Server_Setup.cpp
Client_Options.cpp          ORB_Shutdown.cpp         RTE_Server_Setup.h
Client_Options.h            ORB_Shutdown.h            RTE_Server_Setup.inl
Client_Pair.cpp             ORB_Shutdown.inl          Send_Task.cpp
Client_Pair.h               ORB_Task_Activator.cpp    Send_Task.h
Client_Pair.inl             ORB_Task_Activator.h       Send_Task_Stopper.cpp
Consumer.cpp                ORB_Task.cpp              Send_Task_Stopper.inl
Consumer.h                  ORB_Task.h                Servant_var.cpp
The following mpc file (RTEC_Perf.mpc) shows the simple and small number of lines required to generate usable build tool project files.

```makefile
project(RTEC_Perf): strategies, rtkorbaevent, minimum_corba {
    sharedname = TAO_RTEC_Perf
    idlflags  += -Wb,export_macro=TAO_RTEC_Perf_Export \ 
                 -Wb,export_include=rtec_perf_export.h
    dllflags  += TAO_RTEC_PERF_BUILD_DLL

    Template_Files {
        Auto_Disconnect.cpp
        Auto_Functor.cpp
        Low_Priority_Setup.cpp
        RIR_Narrow.cpp
        Servant_var.cpp
        Shutdown.cpp
        Task_Activator.cpp
    }
}
```

A line-by-line explanation of the example mpc file is listed below.

The first line declares a project named RTEC_Perf that inherits from the base projects listed after the colon.

```makefile
sharedname = TAO_RTEC_Perf
```
Line 2 determines that the project is a library and the library name is
TAO_RTEC_Perf.

```
idlflags  += -Wb,export_macro=TAO_RTEC_Perf_Export \ 
           -Wb,export_include=rtec_perf_export.h
```

Lines 3-4 add to the flags passed to the IDL compiler when processing the idl
files.

```
dllflags  += TAO_RTEC_PERF_BUILD_DLL
```

The next line adds TAO_RTEC_PERF_BUILD_DLL to the dllflags, which
defines a macro that is used by the rtec_perf_export.h header file.

```
Template_Files {
  Auto_Disconnect.cpp
  Auto_Functor.cpp
  Low_Priority_Setup.cpp
  RIR_Narrow.cpp
  Servant_var.cpp
  Shutdown.cpp
  Task_Activator.cpp
}
```

Lines 7-15 name the listed cpp files as part of the Template_Files.

You may have noticed that there isn’t much to the file above. With the default
behaviors that are built into MPC, there does not need to be. We rely on the
defaults to determine the values of IDL_Files, Source_Files,
Inline_Files, and Header_Files. Since the template files do not match
the MPC built-in defaults, we must explicitly list them. We also rely on
inheritance to get many of the TAO-related options.

### 4.4 Adding a New Type

If MPC does not support a particular build tool, you may want to consider
adding a new project type. For instance, support could be added to MPC for
Boost Jam, Eclipse, Xcode and many others. To do so will require knowledge
of the MPC input files, as well as Object Oriented Perl.
4.4.1 Input File Syntax
This section describes the syntax of the files that are used during project generation.

4.4.1.1 Template Files (mpd)
Template files make up the bulk of what MPC puts into each generated project file. They provide the plain text and the layout of the data provided by the mpc files, using various template directives.

Template directives are declared using a `<% %>` construct. This construct is used to create if statements, for loops and to access variables. One thing to note is that any text, including white space, that is not enclosed within `<% %>` is left untouched and is passed directly into the generated project file.

An if statement can appear on a single line or it can span multiple lines. For example, the following line:

```plaintext
<%if(exename)%>BIN = <%exename%><%else%>LIB = <%sharedname%><%endif%
```

is equivalent to:

```plaintext
<%if(exename)%>
  BIN = <%exename%
<%else%
  LIB = <%sharedname%
<%endif%
```

A foreach statement can also appear on a single line or can span multiple lines. As described below in the keywords section, the foreach statement evaluates the variable in a space-separated list context.

There are a couple of ways to write a foreach loop. The first and preferred way is to name the loop variable and then list each variable to be evaluated.

```plaintext
FILES=<%foreach(fvar, idl_files source_files header_files)%> <%fvar%><%endfor%
```

The second way is to let the foreach statement determine the loop variable. With this style, each value can be accessed via the first variable name passed to the foreach with the trailing ‘s’ removed.

```plaintext
FILES=<%foreach(idl_files source_files header_files)%> <%idl_file%><%endfor%
```
4.4 Adding a New Type

Note that the `<%idl_file%>` variable will contain each individual value of the `idl_files`, `source_files` and `header_files` list. If the variable in the `foreach` does not end in 's', the variable of the same name within the `foreach` will contain each individual value, e.g.,

```%
foreach(filelist)<%filelist%>%endfor%
```

Table 4-8 lists keywords that can appear in template files.

### Table 4-8 Template File Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if</strong></td>
<td>Used to determine if a variable is defined. The not operator (!) can be used to invert the if check. This construct will only check for values defined within an mpc or mpt file. Default values (even those implemented by the project creators) are not considered in the if statement.</td>
</tr>
<tr>
<td><strong>else</strong></td>
<td>Used with the if statement. An else block will be evaluated if the statement does not evaluate to true.</td>
</tr>
<tr>
<td><strong>endif</strong></td>
<td>Used with the if statement. This ends an if or if/else block.</td>
</tr>
<tr>
<td><strong>noextension</strong></td>
<td>Evaluates the variable name value as a file name and removes the extension from that value including the period.</td>
</tr>
<tr>
<td><strong>dirname</strong></td>
<td>Evaluates the variable name and removes the basename from that value.</td>
</tr>
<tr>
<td><strong>basename</strong></td>
<td>Evaluates the variable name and removes the directory portion from that value.</td>
</tr>
<tr>
<td><strong>basenoextension</strong></td>
<td>This is similar to basename except that the extension is also removed from the variable name value.</td>
</tr>
<tr>
<td><strong>foreach</strong></td>
<td>The given variable names are evaluated in a list context which is space separated.</td>
</tr>
<tr>
<td><strong>forfirst</strong></td>
<td>Used with foreach. The literal value passed to forfirst will be placed on the first iteration of foreach.</td>
</tr>
<tr>
<td><strong>fornotfirst</strong></td>
<td>Used with foreach. The literal value passed to fornotfirst will be placed on each iteration of foreach except for the first.</td>
</tr>
<tr>
<td><strong>forlast</strong></td>
<td>Used with foreach. The literal value passed to forlast will be placed on the last iteration of foreach.</td>
</tr>
</tbody>
</table>
### Table 4-8 Template File Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fornotlast</td>
<td>Used with foreach. The literal value passed to fornotlast will be placed on each iteration of foreach except for the last.</td>
</tr>
<tr>
<td>endfor</td>
<td>Used with foreach. This ends foreach block.</td>
</tr>
<tr>
<td>comment</td>
<td>The value passed to comment is ignored and can be any set of characters, except a new line or a closing parenthesis.</td>
</tr>
<tr>
<td>flag_overrides</td>
<td>This is directly related to overriding the project-wide settings in an mpc file. It takes two variable names that are comma separated. The first corresponds to a file name and the second is any variable name.</td>
</tr>
<tr>
<td>marker</td>
<td>This is directly related to the verbatim keyword from the mpc syntax. This can be used to designate markers within a template. Ex. <code>&lt;%marker(local)%&gt;</code>.</td>
</tr>
<tr>
<td>uc</td>
<td>Return the given variable value in all upper case characters.</td>
</tr>
<tr>
<td>lc</td>
<td>Return the given variable value in all lower case characters.</td>
</tr>
<tr>
<td>ucw</td>
<td>Return the given variable value with the first letter of each word in upper case. Words are separated by spaces or underscores.</td>
</tr>
<tr>
<td>normalize</td>
<td>Convert dashes, slashes, dollar signs, parenthesis and dots in the given variable value to underscores.</td>
</tr>
<tr>
<td>reverse</td>
<td>This function reverses the order of the array parameter values.</td>
</tr>
<tr>
<td>sort</td>
<td>This function sorts the array parameter values.</td>
</tr>
<tr>
<td>uniq</td>
<td>This function returns the unique set of the array parameter values.</td>
</tr>
<tr>
<td>multiple</td>
<td>This function returns true if the array parameter contains multiple values.</td>
</tr>
<tr>
<td>starts_with</td>
<td>This function returns true if the variable value (first parameter) starts with the regular expression (second parameter).</td>
</tr>
<tr>
<td>ends_with</td>
<td>This function returns true if the variable value (first parameter) ends with the regular expression (second parameter).</td>
</tr>
</tbody>
</table>
4.4 Adding a New Type

Table 4-9 lists special names that can be used as variables in some template files. The variables listed in Table 4 on page 59 can be used as well (except for `<%temporary%>`).

Table 4-9 Special Values used in Template Files

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>custom_types</td>
<td>Contains a list of the custom build types. See “Custom Types” on page 70 for more details.</td>
</tr>
<tr>
<td>cwd</td>
<td>The full current working directory.</td>
</tr>
<tr>
<td>forcount</td>
<td>This only has a value within the context of a foreach and provides a 1 based count of the index of the elements in foreach.</td>
</tr>
<tr>
<td>project_name</td>
<td>This variable contains the name of the current project being generated.</td>
</tr>
<tr>
<td>project_file</td>
<td>This variable contains the name of the output file for the current project being generated.</td>
</tr>
<tr>
<td>ciao</td>
<td>Implemented by the GNUACE project creator module, specifies that the project uses CIAO.</td>
</tr>
<tr>
<td>cppdir</td>
<td>This value is implemented by the BMake project creator modules. It returns a semicolon separated list of directories taken from each value in the Source_Files list.</td>
</tr>
<tr>
<td>rcdir</td>
<td>This value is implemented by the BMake project creator modules. It returns a semicolon separated list of directories taken from each value in the Resource_Files list.</td>
</tr>
<tr>
<td>make_file_name</td>
<td>This value is implemented by the VC6 and EM3 project creator modules. It returns the project name with the make file extension that corresponds to the particular project type.</td>
</tr>
<tr>
<td>tao</td>
<td>Implemented by the GNUACE project creator module, specifies that the project uses TAO.</td>
</tr>
<tr>
<td>guid</td>
<td>This value is implemented by the VC7 project creator module. It returns a guid value based on the project that is usable within VC7 project files.</td>
</tr>
</tbody>
</table>
The Makefile, Project, and Workspace Creator (MPC)

Custom Types
To support multiple custom build types, a special keyword was introduced. The `custom_types` keyword is used to access the list of custom types defined by the user. In a `foreach` context, each custom type can be accessed through the `custom_type` keyword.

A variety of information is available from each `custom_type` through the `->` operator. The input files, input extensions, command, command output option, command flags, and output file directory are all accessible through the field names that correspond to the particular type.

The input files associated with the custom type are accessed through `custom_type->input_files`. Each input file has a set of output files associated with it which can be accessed in a `foreach` context through `custom_type->input_file->output_files`. The custom type fields are listed in Table 4-10.

Table 4-10 Custom Type Fields

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>The command used for the custom type.</td>
</tr>
<tr>
<td>commandflags</td>
<td>The command options not including the output option.</td>
</tr>
<tr>
<td>dependent</td>
<td>This setting determines the command upon which custom generated files should depend.</td>
</tr>
<tr>
<td>gendir</td>
<td>The output directory associated with a particular input file. This field has no meaning when accessed directly through the <code>custom_type</code>. It should always be used within the context of a <code>flag_overrides</code> (See Table 4-8).</td>
</tr>
<tr>
<td>input_files</td>
<td>The input files associated with the custom type.</td>
</tr>
<tr>
<td>inputexts</td>
<td>The input file extensions associated with the custom type.</td>
</tr>
<tr>
<td>libpath</td>
<td>The library path setting for the command.</td>
</tr>
</tbody>
</table>
4.4 Adding a New Type

The example below, which creates generic makefile rules for building custom input files, shows basic use of the custom type and the various fields that can be accessed. The main limitation with the `custom_types` keyword, as can be seen below, is that the `foreach` variable cannot be named as stated on page 66.

```%
<%if(custom_types)%>
<%foreach(custom_types)%>
<%foreach(custom_type->input_files)%>
<%foreach(custom_type->input_file->output_files)%>
<%custom_type->input_file->output_file%>: <%custom_type->input_file%>
    <%custom_type->command%> <%custom_type->commandflags%> $@
<%endfor%>
<%endfor%>
<%endfor%>
<%endif%>
```

Grouped Files

File grouping is part of the syntax of mpc files. If a set of files are grouped within the mpc file, they can be accessed as a group within the mpd file.

Files (such as `Source_Files`, `Header_Files`) can be grouped together as shown on page 50. Within the mpd file, the different components can be accessed by prepending `grouped_` to the component (grouped_source_files, grouped_header_files, etc.)

Table 4-11 Grouped Files Field Names

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>files</td>
<td>The input files associated with the group.</td>
</tr>
<tr>
<td>component_name</td>
<td>The name of the set of multiple groups of files.</td>
</tr>
</tbody>
</table>
The example below, which creates make macros for each file group, shows basic use of grouping and the fields that can be accessed. The main limitation with file grouping, as can be seen below, is that the \texttt{foreach} variable cannot be named as stated on page 66. The following example involves source files, but any of the components listed in 4.3.2.2 can be used.

\begin{verbatim}
<%if(grouped_source_files)%>
<%comment(Get back each set of grouped files)%>
<%foreach(grouped_source_files)%>
<%comment(This will provide the name of the group)%>
<%grouped_source_file%> = \\
<%comment(Get all the source files in a single group)%>
<%foreach(grouped_source_file->files)%>
<%grouped_source_file->file)%><%fornotlast(" ")%>
<%endfor%>
<%endfor%>
ifndef <%grouped_source_files->component_name%>
  <%grouped_source_files->component_name%> = \\
<%foreach(grouped_source_files)%>
  <%grouped_source_file%><%fornotlast(" ")%>
<%endfor%>
endif
<%endif%
\end{verbatim}

4.4.1.2 Template Input Files (mpt)

Template input files provide build tool specific information that is common to all projects, such as compiler switches, intermediate directories, compiler macros, etc. Each project type can provide template input files for dynamic libraries, static libraries, dynamic executables and static executables. However, none of these are actually required by MPC.

The template input files are more free-form than the other MPC file types. It is similar to the mpc syntax except that there is no project definition and there is only one keyword. The keyword, \texttt{conditional\_include}, is used to include other mpt files if they can be found in the MPC include search path. If the name listed in double quotes after \texttt{conditional\_include} is not found, it is ignored and no warning is produced. The \texttt{mpt} extension is automatically added to the name provided.

The template input files contain variable assignments and collections of variable assignments. A variable assignment is of the form:
variable_name = value1 "value 2"
variable_name += another_value

This variable can then be used within the corresponding mpd file.
Variable assignments can be grouped together and named within the mpt file and used as scoped variables within the mpd file. The following example shows the use of collections of variable assignments.

// mpt file
customizations = Release Debug
customizations defines = WIN32 _CONSOLE

Release {
    compile flags = /W3 /GX /O2 /MD /GR
    defines = NDEBUG
}

Debug {
    compile flags = /W3 /Gm /GX /Zi /Od /MDd /GR /Gy
    defines = _DEBUG
}

conditional include "vcfullmacros"

Below is the portion of the mpd file that would use the information provided in the mpt file above.

<%foreach(customizations)%>
Name = <%configuration%>
<%compile_flags%><%foreach(defines customizations defines)%> /D <%define%>=1<%endfor%>
<%endfor%>

The following output is generated from the above example:

Name = Release
/W3 /GX /O2 /MD /GR /D NDEBUG=1 /D WIN32=1 /D _CONSOLE=1

Name = Debug
/W3 /Gm /GX /Zi /Od /MDd /GR /Gy /D _DEBUG=1 /D WIN32=1 /D _CONSOLE=1

If a foreach variable value corresponds to a variable group name, that variable group is available within the scope of that foreach.
4.4.2 A Simple Example

We will discuss what it would take to add support for a fictional build tool throughout this section. The diagram on page 35 shows the relationship between the template and project creator discussed below.

4.4.2.1 Template

The best thing to do is to start with the template. The template is the most important piece when adding a new project type. It basically tells MPC how to lay out all of the information it gathers while processing an mpc file. The template file will have a mixture of plain text and the mpd syntax described in 4.4.1.1. Here is our sample fictional.mpd.

```plaintext
//=======================================================================
// This project has been generated by MPC.
// CAUTION! Hand edit only if you know what you are doing!
//=======================================================================

// Section 1 - PROJECT OPTIONS
ctags:*
debugSwitches:-nw
//end-proj-opts

// Section 2 - MAKEFILE
Makefile.<%project_name%>

// Section 3 - OPTIONS
//end-options

// Section 4  - TARGET FILE
<%if(exename)%>  
<%exename%>
<%else%
<%if(sharedname)%>  
<%sharedname%>
<%else%
<%if(staticname)%>  
<%staticname%>
<%endif%>  
<%endif>
<%endif>

// Section 5  - SOURCE FILES
<%foreach(source_files)%>
<%source_file%>
<%endfor%
//end-srcfiles
```
// Section 6 - INCLUDE DIRECTORIES
<%foreach(includes)%>
<%include%>
<%endfor%>
//end-include-dirs

// Section 7 - LIBRARY DIRECTORIES
<%foreach(libpaths)%>
<%libpath%>
<%endfor%>
//end-library-dirs

// Section 8 - DEFINITIONS
<%foreach(macros defines)%>
-D<%macro%>
<%endfor%>
<%if(pch_header)%>
<%foreach(pch_defines)%>
-D<%pch_define%>
<%endfor%>
<%endif%>
//end-defs

// Section 9 - C FLAGS
<%cflags("-g")%>

// Section 10 - LIBRARY FLAGS
<%libflags%>

// Section 11 - SRC DIRECTORY
.

// Section 12 - OBJ DIRECTORY
<%objdir(".")%>

// Section 13 - BIN DIRECTORY
<%if(install)%%install%%else%>.<%endif%

// User targets section. Following lines will be
// inserted into Makefile right after the generated cleanall target.
// The Project File editor does not edit these lines - edit the .vpj
// directly. You should know what you are doing.
// Section 14 - USER TARGETS
<%marker(top)%%marker(macros)%%marker(local)%%marker(bottom)%>
//end-user-targets

// Section 15 - LIBRARY FILES
Note that output is generated differently depending upon whether `<%exename%>`, `<%sharedname%>` or `<%staticname%>` is defined due to the if statements that were used with relation these variable names. Also, certain portions of the project file are only generated if particular variables are set.

### 4.4.2.2 Project Creator

Next, you would write the `FictionalProjectCreator.pm`. It may be best to start with a copy of the `MakeProjectCreator.pm` and edit it. Change the package name to `FictionalProjectCreator` and have it inherit from `MakeProjectBase` and `ProjectCreator`. Then, override the methods that are needed for this particular type.

```perl
package FictionalProjectCreator;

# *****************************************************
# Description   : A Fictional Project Creator
# Author        : Chad Elliott
# Create Date   : 10/01/2004
# *****************************************************

# *****************************************************
# Pragmas
# *****************************************************
use strict;

use MakeProjectBase;
use ProjectCreator;

use vars qw(@ISA);
@ISA = qw(MakeProjectBase ProjectCreator);

# *****************************************************
# Subroutine Section
# *****************************************************
sub convert_slashes {
    my($self) = shift;
    return 0;
}
```
In our example, we inherit from the MakeProjectBase which provides some methods that are common to all “make” based project creators.

We override the `convert_slashes` method to return 0. A zero return value tells MPC not to convert slashes to back slashes (converting slashes is useful for Windows related build tools).

We then override the `project_file_extension` method to return the project file extension which is used by a method defined in the MakeProjectBase module.

Next, we override the `get_dll_exe_template_input_file` and `get_dll_template_input_file` methods. Those methods return the specific template input file names for a dynamic executable and dynamic library, respectively.

Lastly, we override the `get_template` method to return the template file name for our new project type. In our case, the method returns `fictional` which corresponds to the name of the template file we created earlier.
There are many other methods that can be overridden to change the way MPC generates output. For a complete list, see the “Virtual Methods To Be Overridden” section of the Creator.pm and ProjectCreator.pm.

### 4.4.2.3 Workspace Creator

The next part that you would need to write is the FictionalWorkspaceCreator.pm. This module is usually more code-intensive than its Project Creator counterpart.

```perl
package FictionalWorkspaceCreator;

# ************************************************************
# Description   : A Fictional Workspace Creator
# Author        : Chad Elliott
# Create Date   : 10/01/2004
# ************************************************************

# Pragmas
# ************************************************************
use strict;
use FictionalProjectCreator;
use WorkspaceCreator;

use vars qw(@ISA);
@ISA = qw(WorkspaceCreator);

# ************************************************************
# Subroutine Section
# ************************************************************
sub workspace_file_name {
    my($self) = shift;
    return $self->get_modified_workspace_name($self->get_workspace_name(), '.fws');
}

sub pre_workspace {
    my($self) = shift;
    my($fh) = shift;
    my($crlf) = $self->crlf();

    print $fh '
        <!-- MPC Command -->', $crlf,
        '<!-- $0 @ARGV -->', $crlf,
        '<!-- $0 @ARGV -->', $crlf;
    }
```
4.4 Adding a New Type

The first method we override from WorkspaceCreator.pm is the workspace_file_name method. It is used to determine the output file for the generated workspace.

Second, we override the pre_workspace method, which we use to print out the generic unchanging section of our generated workspace.

Lastly, we override the write_comps method. This method is where the bulk of the work is done in our workspace creator. A workspace creator has many sets of data available. A reference to the list of project file names can be obtained through the get_projects method; project-specific information can be obtained through the get_project_info method which returns an array reference where each array element is an array containing the project name, project dependencies and a project guid (if applicable).

4.4.2.4 MPC.pm and MWC.pm

Perhaps the easiest part of adding a new workspace and project type is modifying the MPC.pm and MWC.pm modules. Each file contains a “Subroutine Section” with a method named “new”. This method contains an array of strings named creators. The creators array contains the names of the available project creators (MPC.pm) and workspace creators (MWC.pm). The only thing that needs to be done for our new type is to add FictionalProjectCreator to the creators list in MPC.pm and
FictionalWorkspaceCreator to the creators list in MWC.pm. It is important that the new type is not added to the beginning of the array. The first type in the array determines the default type and should not be changed. The options will automatically be updated to reflect the new type.
Part 2

Features of TAO
CHAPTER 5

TAO IDL Compiler

5.1 Introduction

To use IDL interfaces with the static invocation approach, you must generate skeleton and stub C++ code so requests can traverse from a client to a servant. TAO includes an IDL compiler, tao_idl, that generates C++ skeletons and stubs from your IDL file.

The generated code is only usable by TAO. The output from IDL-to-C++ compilers cannot be interchanged among CORBA implementations. However, the code generated by TAO’s IDL compiler is platform-independent, making it possible to use TAO in cross-compilation environments.

TAO’s IDL compiler maps IDL files to C++ according to the CORBA C++ mapping specification (OMG Document formal/03-06-03). The basic C++ mapping uses C++ exceptions to report system and user exceptions. An alternate mapping for environments that do not use native C++ exceptions allows the use of CORBA::Environment variables to pass error information via operation parameters. The TAO IDL compiler supports both mappings.
See 5.9 and 6.4 for more information on using the alternate C++ mapping for exceptions with TAO.

The IDL compiler is modularized into a top-level executable, plus libraries for the front- and back-ends. This modular design allows the front-end lexing and parsing engine to be reused and different back-ends to be “plugged in” to produce different outputs (e.g., to populate the Interface Repository).

5.2 Executables

UNIX and UNIX-like Systems

The IDL compiler executable is $TAO_ROOT/TAO_IDL/tao_idl, with a symbolic link in $ACE_ROOT/bin.

Windows Systems

The IDL compiler executable is %ACE_ROOT%\bin\tao_idl.exe.

General Usage

The general usage of the TAO IDL compiler is as follows:

```
tao_idl <options> IDL-file(s)
```

IDL file names must be listed after the options. The options will apply to all of the IDL files. For example:

```
tao_idl -GI -Ge 1 hello.idl Messenger.idl
```

5.3 Output Files Generated

By default, nine files are generated for every IDL file the tao_idl compiler processes. Three of these files provide the stub code used by the client, and six files provide the skeleton code used by the server. The generation of these files ensures that the generated code is portable and optimized for a wide variety of C++ compilers. However, your client and server applications only need to include two header files directly.
5.3 Output Files Generated

For an IDL file named Messenger.idl, running the command

```
tao_idl Messenger.idl
```
generates the following files (we show how to customize these names later):

**Table 5-1 C++ Files Generated**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessengerC.h</td>
<td>Stub class definitions.</td>
</tr>
<tr>
<td>MessengerC.inl</td>
<td>Inline stub member function definitions.</td>
</tr>
<tr>
<td>MessengerC.cpp</td>
<td>Stub member function definitions.</td>
</tr>
<tr>
<td>MessengerS.h</td>
<td>Skeleton class definitions.</td>
</tr>
<tr>
<td>MessengerS.inl</td>
<td>Inline skeleton member function definitions.</td>
</tr>
<tr>
<td>MessengerS.cpp</td>
<td>Skeleton member function definitions.</td>
</tr>
<tr>
<td>MessengerS_T.h</td>
<td>Skeleton class template definitions.</td>
</tr>
<tr>
<td>MessengerS_T.inl</td>
<td>Inline skeleton template member function definitions.</td>
</tr>
<tr>
<td>MessengerS_T.cpp</td>
<td>Skeleton template member function definitions.</td>
</tr>
</tbody>
</table>

### 5.3.1 Tips for Working with the Output Files

- The client includes `MessengerC.h` and links to `MessengerC.o`. The server includes `MessengerS.h` and links to both `MessengerC.o` and `MessengerS.o`.

- The stubs and skeletons are decoupled. The client implementation need not include the `*S.h` file or link with `*S.o`. However, the server requires knowledge of the stub classes and IDL types found in the `*C.h` files. Therefore, the generated `*S.h` files that you include in your server code include the corresponding `*C.h` files. In addition, your server implementation needs to link with the object code produced from both the `*C.cpp` and `*S.cpp` files.

- The template code in the files ending in `S_T.*` is included in the `*S.*` files. Therefore, you do not need to explicitly refer to them in your code or build tools.

- The compiler interprets the functions in the `*.inl` files as inline code only if the build defines the preprocessor macro `__ACE_INLINE__`. This makes it possible to build applications with inlining disabled (e.g., to
facilitate debugging) or with inlining enabled (e.g., to improve performance). You can find the definition of this preprocessor option by looking for the \_ACE\_INLINE\_ macro in 
$ACE\_ROOT/ace/config.h.

5.4 Using TAO IDL Compiler Options

We discuss command line options available with the IDL compiler in 5.5 through 5.14. To see a complete list of the IDL compiler’s options, enter the following:

\texttt{tao\_idl -u}

The file \texttt{$TAO\_ROOT/docs/compiler.html} also contains information on IDL compiler options.

You can specify IDL compiler options directly on the command line, or you can specify them in an MPC file by adding your IDL compiler options to the idlflags keyword. For example:

\texttt{idlflags += -I$(TAO\_ROOT)/orbsvcs -Ge 0}

5.5 Preprocessing Options

As required by the CORBA specification, IDL files can contain directives defined by the C++ preprocessor. This means, for example, that one IDL file can include another IDL file by using the \#include directive. Likewise, conditional compilation can be done by using the \#ifdef or \#if defined directives. TAO’s IDL compiler does not include a preprocessor, rather it invokes an external preprocessor. By default, the preprocessor is specified by the \texttt{TAO\_IDL\_PREPROCESSOR} environment variable. This variable is defined when \texttt{tao\_idl} is built. The C++ compiler’s preprocessor is used if this variable is not defined. See 5.5.1 for information on how to override the default and specify the preprocessor during IDL compilation.

Some common preprocessor options can be passed to the IDL compiler, which will then pass them through to the preprocessor. The IDL compiler also
5.5 Preprocessing Options

supports the passing of any option to the preprocessor via its \texttt{-Wp} option. Table 5-2 provides details of the options related to preprocessing.

**Table 5-2 Preprocessing Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{-E}</td>
<td>Run the preprocessor on the IDL file, but do not generate any C++ code.</td>
<td>\texttt{tao_idl -E Messenger.idl}</td>
</tr>
<tr>
<td>\texttt{-D \textit{macro-def}}</td>
<td>Defines a macro.</td>
<td>\texttt{tao_idl -D CORBA_IMPL=tao Messenger.idl}</td>
</tr>
<tr>
<td>\texttt{-U \textit{macro-def}}</td>
<td>Undefines a macro.</td>
<td>\texttt{tao_idl -U unix Messenger.idl}</td>
</tr>
<tr>
<td>\texttt{-I \textit{include-path}}</td>
<td>Add \textit{include-path} to the list of paths searched for include files.</td>
<td>\texttt{tao_idl -I /idl/exceptions -I $TAO_ROOT/orbsvcs/orbsvcs Messenger.idl}</td>
</tr>
<tr>
<td>\texttt{-A \textit{assertion}}</td>
<td>Make an assertion.</td>
<td>\texttt{-A system(gnu)}</td>
</tr>
<tr>
<td>\texttt{-Yp, \textit{preproc-loc}}</td>
<td>Tells the TAO IDL compiler to use a specific preprocessor.</td>
<td>\texttt{tao_idl -Yp, /usr/bin/cpp Messenger.idl}</td>
</tr>
<tr>
<td>\texttt{-Wp, \textit{arg1,arg2,...}}</td>
<td>Passes arguments to the preprocessor.</td>
<td>\texttt{-Wp, -undef}</td>
</tr>
</tbody>
</table>

The \texttt{-D}, \texttt{-U}, \texttt{-I}, and \texttt{-A} options are all passed directly through to the preprocessor. If the preprocessor you are using is the same as your C++ preprocessor (the likely case), then you should see your C++ compiler documentation for details about these options. The \texttt{-Wp} option will only pass the text between the commas to the preprocessor (stripping off the leading \"-Wp\" and all commas).

In addition to accepting preprocessor directives such as \texttt{#define}, \texttt{#include}, and \texttt{#if}, TAO’s IDL compiler recognizes and handles the following preprocessor directives:

**Table 5-3 Additional Preprocessor Directives**

<table>
<thead>
<tr>
<th>Directive</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{#file &quot;file name&quot;}</td>
<td>Identifies the name of the file being preprocessed. These directives are inserted by some preprocessors. The IDL compiler simply ignores them.</td>
<td>\texttt{#file &quot;Messenger.idl&quot;}</td>
</tr>
</tbody>
</table>
5.5.1 Environment Variables Affecting Preprocessing

Environment variables that impact the preprocessing stage of the IDL compiler are described in Table 5-4.

Table 5-4 Preprocessing Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
<th>Default Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAO_IDL_PREPROCESSOR</td>
<td>Specifies the command to access the C++ preprocessor.</td>
<td>Uses the preprocessor used to build the IDL compiler itself.</td>
</tr>
<tr>
<td>TAO_IDL_PREPROCESSOR_ARGS</td>
<td>Provides additional options for the IDL compiler to pass to the preprocessor.</td>
<td>Always passes -DIDL and -I, in addition to any specified options.</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>If set, append its contents to the include path.</td>
<td>Not passed.</td>
</tr>
<tr>
<td>TAO_ROOT</td>
<td>If set, passes -I$(TAO_ROOT)/tao.</td>
<td>Not passed.</td>
</tr>
<tr>
<td>ACE_ROOT</td>
<td>If set, passes -I$(ACE_ROOT)/TAO/tao.</td>
<td>Not passed.</td>
</tr>
</tbody>
</table>

Either ACE_ROOT or TAO_ROOT must be defined for tao_idl to find the file orb.idl when it is included by another IDL file (e.g., #include <orb.idl>). If neither ACE_ROOT nor TAO_ROOT is defined, the IDL compiler will display a warning message.
5.6 Output File Options

The TAO IDL compiler has an option that allows you to specify a target directory for the output files. In addition, there are options to control the file names of the C++ source code generated. These file names always start with the base name of the IDL file being processed. You can change the suffix of the file names that are generated by using the options listed in Table 5-5.

Table 5-5 Output File Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o output-directory</td>
<td>Subdirectory in which to place the generated stub and skeleton files.</td>
<td>Current directory</td>
</tr>
<tr>
<td>-hc filename-ending</td>
<td>Stub header suffix.</td>
<td>C.h</td>
</tr>
<tr>
<td>-ci filename-ending</td>
<td>Stub inline functions suffix.</td>
<td>C.inl</td>
</tr>
<tr>
<td>-cs filename-ending</td>
<td>Stub non-inline functions suffix.</td>
<td>C.cpp</td>
</tr>
<tr>
<td>-hs filename-ending</td>
<td>Skeleton header file name suffix.</td>
<td>S.h</td>
</tr>
<tr>
<td>-si filename-ending</td>
<td>Skeleton inline functions file name suffix.</td>
<td>S.inl</td>
</tr>
<tr>
<td>-ss filename-ending</td>
<td>Skeleton non-inline functions file name suffix.</td>
<td>S.cpp</td>
</tr>
<tr>
<td>-hT filename-ending</td>
<td>Skeleton template header file name suffix.</td>
<td>S_T.h</td>
</tr>
<tr>
<td>-st filename-ending</td>
<td>Skeleton template inline functions file name suffix.</td>
<td>S_T.inl</td>
</tr>
<tr>
<td>-sT filename-ending</td>
<td>Skeleton template non-inline functions file name suffix.</td>
<td>S_T.cpp</td>
</tr>
<tr>
<td>-hI filename-ending</td>
<td>Starter implementation header file name suffix (use with -GI).</td>
<td>I.h</td>
</tr>
<tr>
<td>-sI filename-ending</td>
<td>Starter implementation file name suffix (use with -GI).</td>
<td>I.cpp</td>
</tr>
</tbody>
</table>

As an example of these options, suppose you are migrating from a different CORBA implementation to TAO. When processing an IDL file named Messenger.idl, this implementation generates files named Messenger.hh and Messenger.cc for the stub code, and files named MessengerS.hh and MessengerS.cc for the skeleton code.

For TAO to emulate these naming conventions, invoke the IDL compiler as:

```
 tao_idl -hc .hh -ci .i -cs .cc -hs S.hh -si S.i -ss S.cc -hT S_T.hh -st S_T.i -sT S_T.cc Messenger.idl
```
This will produce header and source file names consistent with the other CORBA implementation.

5.7 Starter Implementation Files

To help you start writing implementation code, `tao_idl` optionally generates starter servant implementation files. The starter files contain empty C++ member function definitions that you must fill in with your implementation code. If you are new to CORBA or have a lot of operations to implement, this can be a great time saver. Table 5-6 lists the options related to this feature.

**Note**  Running the IDL compiler with the starter implementation options overwrites any existing implementation files of the same names. Any modifications will be lost unless you rename the starter implementation files after they are generated (recommended)!

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-GI</code></td>
<td>Generate starter implementation code.</td>
<td>Do not generate starter implementation code.</td>
</tr>
<tr>
<td><code>-GIh file-ending</code></td>
<td>Sets the starter implementation header file suffix to <code>file-ending</code>.</td>
<td><code>I.h</code></td>
</tr>
<tr>
<td><code>-GIS file-ending</code></td>
<td>Sets the starter implementation source file suffix to <code>file-ending</code>.</td>
<td><code>I.cpp</code></td>
</tr>
<tr>
<td><code>-GIb class-prefix</code></td>
<td>Prefix with which to begin the starter implementation class names.</td>
<td>No default prefix.</td>
</tr>
<tr>
<td><code>-GIe class-prefix</code></td>
<td>Suffix with which to end the starter implementation class names.</td>
<td><code>_i</code></td>
</tr>
<tr>
<td><code>-Glc</code></td>
<td>Create an empty copy constructor.</td>
<td>No copy constructor defined.</td>
</tr>
<tr>
<td><code>-Glc</code></td>
<td>Create an empty assignment operator.</td>
<td>No assignment operator defined.</td>
</tr>
</tbody>
</table>

**Note**  The `-GIh` and `-GIS` options have the same effect as the `-hI` and `-sI` options presented in Table 5-5.
For example, consider the following:

```c++
// Messenger.idl
interface Messenger {
  boolean send_message(in    string user_name,
                       in    string subject,
                       inout string message);
};
```

Suppose the convention you use includes implementation files that end with `_i.h` and `_i.cpp`. Then invoking the IDL compiler with:

```
tao_idl -GIh _i.h -GIs _i.cpp Messenger.idl
```

creates the `Messenger_i.h` and `Messenger_i.cpp` starter files.

**Note**  *The -GI option was not needed in this case since using -GIh and -GIs implies -GI.*

The generated starter implementations of operations and attributes contain no code, only comments of the form "//Add your implementation here." For example:

```c++
CORBA::Boolean Messenger_i::send_message (const char* user_name,
                                           const char* subject,
                                           char*& message)
ACE_THROW_SPEC ((
  CORBA::SystemException
))
{
  //Add your implementation here
}
```

To implement the interface, search for these comments and replace them with your own code. However, the IDL compiler does not generate comments for constructors and destructors; do not forget to fill in these functions as well.
5.8 Additional Code Generation Options

In addition to generating starter implementation classes, the TAO IDL compiler provides options for generating reply handler classes for use with the Asynchronous Method Invocation (AMI) callback model, servant and response handler classes for use with Asynchronous Method Handling (AMH), smart proxy factory and default smart proxy classes, optimized TypeCodes, and explicit template instantiations. Options for using the above features are shown in Table 5-7. For more information on AMI callbacks, see 7.2. For more information on AMH, see Chapter 8. For more information on smart proxies, see Chapter 12.

Table 5-7 Additional Code Generation Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-GC</td>
<td>Generate AMI callback reply-handler classes.</td>
<td>AMI callback classes not generated.</td>
</tr>
<tr>
<td>-GH</td>
<td>Generate AMH servant and response-handler classes.</td>
<td>AMH classes not generated.</td>
</tr>
<tr>
<td>-Gsp</td>
<td>Generate smart proxy factory and default smart proxy classes.</td>
<td>Smart-proxy-related classes not generated.</td>
</tr>
<tr>
<td>-Gt</td>
<td>Generate optimized TypeCodes.</td>
<td>TypeCodes not optimized.</td>
</tr>
<tr>
<td>-GA</td>
<td>Generate any operators and TypeCodes into a separate *.cpp stub file.</td>
<td>Generate any operators and TypeCodes into the*.cpp stub file</td>
</tr>
<tr>
<td>-GT</td>
<td>Generate explicit template instantiations.</td>
<td>Explicit template instantiations not generated.</td>
</tr>
<tr>
<td>-Guc</td>
<td>If an IDL constant is declared at module scope, assign its value in the stub’s .cpp file rather than inline in the stub’s header file.</td>
<td>A constant with module scope is assigned its value inline in the stub’s header file.</td>
</tr>
<tr>
<td>-Gdpps</td>
<td>Generate appropriate marshaling and instance key support code for DDS DCPS-enabled types (see Chapter 31).</td>
<td>DDS DCPS-related code not generated.</td>
</tr>
<tr>
<td>-in</td>
<td>Generate #includes within &quot;&lt;&gt;&quot;</td>
<td>Uses &quot;&quot; by default.</td>
</tr>
<tr>
<td>-ic</td>
<td>Generate #includes within &quot;&quot;</td>
<td>Uses &quot;&quot; by default.</td>
</tr>
</tbody>
</table>
The TAO IDL compiler can generate C++ code that supports the alternate C++ mapping for systems without native C++ exception handling. For more details on this mapping see the discussion in 6.4 and the corresponding CORBA specification documentation.

To support this mapping, an ACE macro, `ACE_ENV_ARG_DECL`, is appended to the end of each remote operation’s parameter list by the TAO IDL compiler. For example, the IDL operation

```idl
boolean send_message(in    string user_name,
                     in    string subject,
                     inout string message);
```

is mapped to the following C++ member function:

```c++
virtual CORBA::Boolean send_message (const char* user_name,
                                         const char* subject,
                                         char*& message ACE_ENV_ARG_DECL)
ACE_THROW_SPEC (;
    CORBA::SystemException
));
```

Normally, the TAO IDL compiler does not generate these extra parameters. However, when built with compilers/systems that do not support exceptions or when native C++ exceptions are disabled, the compiler generates these parameters by default. Generation of the `ACE_ENV_ARG_DECL` parameters can always be explicitly controlled using the `-Ge` option, as shown in Table 5-8.

**Table 5-8 Exception Handling Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
</table>
| `-Ge (0|1|2)` | 0 = disable environment parameter generation.  
1 = enable environment parameter generation. 
2 = same as 0, plus generates throw in function signatures instead of `ACE_THROW_SPEC`, et al. | If `ACE_HAS_EXCEPTIONS` is defined, the compiler does not generate the environment parameters; otherwise, it does. |

There are four ways to use the TAO IDL compiler to generate the appropriate mapping for exceptions in the stubs and skeletons:
1. You can use it in an environment that does not support C++ exceptions (i.e., where the C++ preprocessor macro `ACE_HAS_EXCEPTIONS` is not defined). In this case the TAO IDL compiler maps the ACE macro `ACE_ENV_ARG_DECL` to the CORBA::Environment parameters.

2. You can use it in an environment that supports C++ exceptions (i.e., where the C++ preprocessor macro `ACE_HAS_EXCEPTIONS` is defined). You may wish to use C++ exceptions in such cases rather than relying on the `ACE_ENV_ARG_DECL` (which maps to the CORBA::Environment) parameters. In these circumstances the TAO IDL compiler will not generate the `ACE_ENV_ARG_DECL` parameters.

3. You may wish to perform a cross-platform compilation for a target that uses a different exception-handling model than what is being used on your current platform. For example, you may be compiling on a platform that supports exceptions, but the generated code may be used on a platform that does not support exceptions.

4. You may have legacy code that uses the CORBA::Environment parameter, and you do not wish to upgrade the code to use C++ exceptions instead.

The compiler can recognize cases (1) and (2), and can generate code accordingly. It would be very difficult for the compiler to identify cases (3) and (4) without user input. For cases (3) and (4), where C++ exceptions are enabled, using the option `-Ge 1` will enable the appropriate environment parameter generation.

---

**Note**  
*Code generated with the `ACE_ENV_ARG_DECL` macro can be used regardless of the platform’s support for native exceptions. It expands to the CORBA::Environment parameter on platforms that do not support native exceptions. On platforms that do support native exceptions, the macro expands to nothing and therefore does not introduce additional code.*

---

**Note**  
*The `-Ge 2` option is useful when using tools that reverse-engineer C++ source code, such as some UML modeling tools and documentation generators. Such tools are often unable to correctly parse and understand the `ACE_THROW_SPEC` and related macros.*
5.10 Operation Lookup Strategy Options

When a server receives a request from a client, the POA needs to find the skeleton function associated with the operation name in the request. This step involves looking up an operation, based on a string contained in the client request. There are many ways to do this; each has its strengths and weaknesses. Table 5-9 shows the operation lookup strategies for which the IDL compiler can generate code. For most cases we recommend that you use the default strategy, perfect hashing, which is usually optimal in both time and space.

Table 5-9 Operation Lookup Strategies

<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
<th>Lookup time</th>
</tr>
</thead>
<tbody>
<tr>
<td>-H perfect_hash</td>
<td>Perfect hashing</td>
<td>Constant. Suitable for hard real-time systems.</td>
</tr>
<tr>
<td>-H linear_search</td>
<td>Linear search</td>
<td>Proportional to the number of operations. Represents a worst-case strategy for comparison purposes.</td>
</tr>
<tr>
<td>-H binary_search</td>
<td>Binary search</td>
<td>Proportional to the log of the number of operations. Adding more operations has minimal impact on lookup time.</td>
</tr>
<tr>
<td>-H dynamic_hash</td>
<td>Dynamic hash</td>
<td>Constant for the average case. Worst case similar to linear search. Inappropriate for hard real-time systems.</td>
</tr>
</tbody>
</table>

To support the perfect hashing operation lookup strategy, the TAO IDL compiler relies on gperf, a general purpose perfect hashing function generator that is a separate application program distributed in source code form with ACE and TAO. It is invoked by the TAO IDL compiler when the perfect hashing operation lookup strategy is selected. The default path for gperf is $ACE_ROOT/bin/gperf. To override this value, use the -g option and specify the full path to the location of the gperf executable. For example, if gperf is installed in /usr/local/bin instead of $ACE_ROOT/bin, you should invoke the IDL compiler as follows:

$TAO_ROOT/TAO_IDL/tao_idl -g /usr/local/bin/gperf Messenger.idl
5.11 Collocation Strategy Options

The use of collocated stubs allows requests on collocated servants to be dispatched more directly by permitting requests to bypass several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic.

TAO provides two strategies for generating and using collocated stubs.

- The *thru_poa* collocation strategy delivers the request through the servant’s POA and is considered the standard collocated stub.
- The *direct* collocation strategy delivers the request directly from the stub to the servant as a normal C++ virtual function call, thereby bypassing the POA completely.

Collocation strategy options are shown in Table 5-10. For more details on using these collocation strategies at run time, see 19.8.3.

**Table 5-10 Collocation Strategy Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Gp (default)</td>
<td>Generate collocated stubs that use <em>thru_poa</em> collocation strategy.</td>
</tr>
<tr>
<td>-Gd</td>
<td>Generate collocated stubs that use <em>direct</em> collocation strategy.</td>
</tr>
</tbody>
</table>

**Note**  
The run-time ORB initialization options that affect collocation must be compatible with the types of generated collocated stubs. For example, using the *-Gd* option at compile time, then using the *-ORBCollocationStrategy thru_poa* option at run time, is inconsistent and results in a run-time exception.

5.12 Back End Options

The *-Wb* option can be used to pass options to the TAO-IDL-compiler back end that generates the C++ code. The general format for these options is as follows:

  

  `-Wb,option_list`
5.12 Back End Options

The option list is a comma-separated list that may contain any of the options shown in Table 5-11. These options mainly control platform-specific behavior of the back end.

On Windows platforms, a header file that defines export macros must be generated. The `%ACE_ROOT\bin\generate_export_file.pl` script generates the header file that is included through `export_include`. For example:

    generate_export_file.pl Messenger > MessengerExport.h

The header file is used by `export_include` as follows:

    tao_idl -Wb,export_macro=Messenger_Export
    -Wb,export_include=MessengerExport.h Messenger.idl

The options ending in `export_macro` trigger the inclusion of these macros, which are necessary on Windows platforms to allow the export of symbols from DLLs. The options ending in `export_include` generate `#include` directives that include the files containing the export macros.

Table 5-11 Back End Options for `-Wb`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>export_macro=macro</code></td>
<td>IDL compiler will emit the macro after each class or extern keyword (both stub and skeleton).</td>
</tr>
<tr>
<td><code>export_include=inc_path</code></td>
<td>IDL compiler will include the file specified by <code>inc_path</code> at the top of the client header.</td>
</tr>
<tr>
<td><code>skel_export_macro=macro</code></td>
<td>IDL compiler will emit the macro after each class or extern keyword in the skeleton code.</td>
</tr>
<tr>
<td><code>skel_export_include=inc_path</code></td>
<td>IDL compiler will include the file specified by <code>inc_path</code> at the top of the server headers.</td>
</tr>
<tr>
<td><code>stub_export_macro=macro</code></td>
<td>IDL compiler will emit the macro after each class or extern keyword in the stub code.</td>
</tr>
<tr>
<td><code>stub_export_include=inc_path</code></td>
<td>IDL compiler will include the file specified by <code>inc_path</code> at the top of the client header.</td>
</tr>
<tr>
<td><code>pch_include=inc_path</code></td>
<td>IDL compiler will include the file specified by <code>inc_path</code> in all generated files (used to support pre-compiled header mechanisms).</td>
</tr>
<tr>
<td><code>pre_include=file</code></td>
<td>IDL compiler will include the file specified by <code>file</code> at the beginning of the generated header file.</td>
</tr>
</tbody>
</table>
Here is an example showing how to use the back end options `export_macro` and `export_include`. This example shows how the IDL compiler is invoked when building the TAO_CosNaming shared library:

```
tao_idl -Wb,pch_include=CosNaming_pch.h -Wb,export_macro=TAO_Naming_Export -Wb,export_include=Naming/naming_export.h -I../.. -I../../orbsvcs -Ge 1 -Sc -Wb,pre_include=ace/pre.h -Wb,post_include=ace/post.h CosNaming.idl
```

### 5.13 Suppression Options

Table 5-12 shows options you can use to suppress the generation of code that corresponds to certain CORBA features particular applications may not need.Suppressing some of the code normally generated for these features may produce smaller skeletons and stubs, which is important for memory-constrained systems.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Sa</code></td>
<td>Suppress generation of the <code>any</code> operators.</td>
<td>The application cannot use the <code>any</code> data type in operation parameter lists.</td>
</tr>
<tr>
<td><code>-St</code></td>
<td>Suppress generation of TypeCodes for IDL-defined types. Automatically implies <code>-Sa</code>.</td>
<td>The application can neither use the <code>any</code> data type in operation parameter lists nor use a TypeCode for any type declared in the IDL file.</td>
</tr>
<tr>
<td><code>-Sp</code></td>
<td>Suppress generation of <code>thru_poa</code> collocated stubs.</td>
<td>Collocation must be disabled or the <code>direct</code> collocation strategy must be used (see 19.5).</td>
</tr>
<tr>
<td><code>-Sd</code></td>
<td>Suppress generation of <code>direct</code> collocation stubs.</td>
<td>Collocation must be disabled or the <code>thru_poa</code> collocation strategy must be used (see 19.5).</td>
</tr>
<tr>
<td><code>-Sc</code></td>
<td>Suppress generation of TIE classes and files.</td>
<td>Application should not use the TIE approach.</td>
</tr>
</tbody>
</table>
5.14 Output and Reporting Options

Table 5-12 Suppression Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Sm</td>
<td>Disable processing of IDL3 constructs.</td>
<td>IDL interfaces cannot use IDL3 constructs (See Chapter 32, the CORBA Component Model).</td>
</tr>
</tbody>
</table>

**Note** The run-time ORB initialization options that affect collocation must be compatible with the types of generated collocated stubs. For example, using the -Sp option at compile time, then using the -ORBCollocationStrategy thru_poa option at run time, is inconsistent and results in a run-time exception.

### 5.14 Output and Reporting Options

Table 5-13 lists options you can use to control the output of various warning, error, and informational messages, as well as the location of temporary files generated by the IDL compiler.

#### Table 5-13 Output and Reporting Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-t dir</td>
<td>Directory used by the IDL compiler for temporary files.</td>
<td>In UNIX, uses the value of the TMPDIR environment variable, if set, or /tmp by default. In Windows, uses the value of the TEMP environment variable, if set, or the WINNT directory (on NT).</td>
</tr>
<tr>
<td>-v</td>
<td>Verbose flag. IDL compiler will print progress messages after completing major phases.</td>
<td>No progress messages displayed.</td>
</tr>
<tr>
<td>-d</td>
<td>Print the Abstract Syntax Tree (AST) to stdout.</td>
<td>AST is not displayed.</td>
</tr>
<tr>
<td>-w</td>
<td>Suppress warnings.</td>
<td>All warnings displayed.</td>
</tr>
<tr>
<td>-V</td>
<td>Print version information for front end and back end.</td>
<td>No version information displayed.</td>
</tr>
<tr>
<td>-Cw</td>
<td>Output a warning if two identifiers in the same scope differ in spelling only by case.</td>
<td>Error output is default.</td>
</tr>
</tbody>
</table>
Table 5-13 Output and Reporting Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Ce</td>
<td>Output an error if two identifiers in the same scope differ in spelling only by case.</td>
<td>Error output is default.</td>
</tr>
</tbody>
</table>
CHAPTER 6

Error Handling

6.1 Introduction

The inherent complexity of distributed applications increases the opportunity for errors to occur. To handle errors, distributed computing middleware needs a mechanism to communicate errors between components. Likewise, clients must be able to handle the error conditions communicated to them by servers.

Distributing an application across several processes and/or several hosts creates more opportunities for errors to occur. Figure 6-1 illustrates a distributed application with several objects distributed across three processes on two hosts. Possible errors include a hardware failure on one of the hosts, the loss of a network connection between the hosts, and a software failure in one of the server processes.
Before getting into TAO specifics, we first summarize the error-handling mechanisms common to all CORBA ORBs. There are two types of errors that may occur in a distributed system:

- **System-level errors**: These error conditions can happen in any distributed system. For example, a server can exit unexpectedly, thereby causing a loss of communication. Likewise, a client may send a request to an object that does not exist.

- **User-level errors**: These are application- and domain-specific error conditions defined by the architects and designers of a distributed system during application design and development. For example, a bank customer may attempt to withdraw more money from her bank account than is in the account.

By default, C++ exceptions are used by the CORBA IDL-to-C++ mapping to communicate error conditions between the client and server components. For more details on error handling in CORBA, see *Advanced CORBA Programming with C++*, 7.15.

Sometimes C++ exceptions are not available or desired. For instance, some platforms and compilers do not support C++ exceptions. Moreover, some applications cannot tolerate the performance impact or increase in code size that using C++ exceptions causes. The OMG defines an alternate mapping for such systems that passes error information through a CORBA::Environment parameter with each invocation. The TAO IDL compiler will automatically generate code based on the appropriate mapping for a given platform, but can
also be directed to generate a specific mapping based on project preferences. See 5.9 for details on the IDL compiler behavior.

TAO uses a series of macros for error handling that allow the code to work properly in both C++ mappings. These macros are useful for C++ applications that desire the same flexibility. The macros also provide more readable code when using the alternate C++ mapping based on the CORBA::Environment parameter. The macros themselves are defined in ACE. For a description of these macros and their use, see 6.5.

6.2 CORBA System Exceptions

By default, CORBA uses C++ exceptions to communicate error conditions between clients and servers. System-level errors automatically communicate with clients through a set of standard CORBA system exceptions. A system exception can be raised during any remote invocation. Table 6-1 lists some common system exceptions. For a complete list of system exceptions, please see Advanced CORBA Programming with C++, 7.15.

Table 6-1 Common System Exceptions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::COMM_FAILURE</td>
<td>The client’s request was accepted, but a failure occurred (e.g., unexpected server termination) while processing the request.</td>
</tr>
<tr>
<td>CORBA::INV_OBJREF</td>
<td>The client attempted to invoke an operation on an invalid object reference.</td>
</tr>
<tr>
<td>CORBA::OBJECT_NOT_EXIST</td>
<td>The client attempted to invoke an operation on a non-existent object (e.g., not activated in the POA).</td>
</tr>
<tr>
<td>CORBA::TRANSIENT</td>
<td>The request was not able to reach its destination because a critical resource needed to carry out the request (e.g., the POA, the server, a connection) was not available.</td>
</tr>
<tr>
<td>CORBA::TIMEOUT</td>
<td>A request could not be completed within the specified time-to-live period as defined by the effective messaging quality of service (QoS) policies.</td>
</tr>
</tbody>
</table>

In C++, all CORBA exceptions derive from the class CORBA::Exception. All system exceptions derive from CORBA::SystemException, as shown in Figure 6-2.
Error Handling

A client must catch system exceptions raised by remote operations. CORBA system exceptions contain information that can aid in debugging.

Table 6-2 lists several operations you can use to get information from a CORBA exception.

Table 6-2 CORBA::Exception Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const char* _rep_id()</td>
<td>Returns the Interface Repository ID of the exception.</td>
</tr>
<tr>
<td>const char* _name()</td>
<td>Returns the name of the exception.</td>
</tr>
<tr>
<td>void _raise()</td>
<td>Throws the exception.</td>
</tr>
<tr>
<td>static CORBA::Exception* _downcast(CORBA::Exception*)</td>
<td>“Downcast” an exception to a more-derived type. (Similar to _narrow() for object references.)</td>
</tr>
</tbody>
</table>

**TAO Extensions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::TypeCode_ptr _type()</td>
<td>Returns the typecode of the exception.</td>
</tr>
<tr>
<td>int _is_a (const char* rep_id)</td>
<td>Returns non-zero if Repository ID of exception matches rep_id.</td>
</tr>
</tbody>
</table>
6.2 CORBA System Exceptions

The following examples illustrate the use of system exceptions in client code. Both of these examples are based on a simple interface of a hotel with guest rooms. In these examples, the client makes the following remote invocations:

- Getting the object reference from the Naming Service.
- Obtaining the hotel name with `hotel->name()`.
- Acquiring a reference to a guest room via `hotel->checkIn()`.
- Obtaining the room number using `room->roomNumber()`.

Any one of these invocations has the potential to raise a system exception.

Here is the IDL for our simple hotel:

```c++
interface GuestRoom
{
    readonly attribute short roomNumber;
    readonly attribute float balance;

    void checkOut();
};

interface Hotel
{
    readonly attribute string name;

    GuestRoom checkIn(in short numNights);
};
```

### Table 6-2 CORBA::Exception Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void _tao_print_exception (const char *info, FILE *f = stdout)</code></td>
<td>Print helpful debugging information about the exception to f, prepended by info.</td>
</tr>
<tr>
<td><code>ACE_CString _info()</code></td>
<td>Returns information printed by <code>_tao_print_exception()</code> as a string.</td>
</tr>
</tbody>
</table>

#### CORBA::SystemException Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::ULong minor()</td>
<td>Returns an ORB-specific error code that conveys additional information about the error. See 6.7 for the TAO minor codes.</td>
</tr>
<tr>
<td>CORBA::CompletionStatus completed()</td>
<td>Returns an enumerated value that indicates whether the operation completed or not before the exception was raised. Valid values are YES, NO, and MAYBE.</td>
</tr>
</tbody>
</table>
Example

Here is an example of a client that catches system exceptions generically. Note that the catch code is highlighted.

```c++
#include <corba.h>
#include <iostream>

int main(int argv, char* argc[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argv, argc);

        Hotel_var hotel = // get a Hotel proxy, possibly through the Naming Service
        CORBA::String_var name = hotel->name();
        std::cout << "The name of the hotel is " << name << std::endl;

        CORBA::Short numNights = 5;
        GuestRoom_var room = hotel->checkIn(numNights);
        std::cout << "The room number is " << room->roomNumber() << std::endl;
        orb->destroy();
    }

    catch (CORBA::SystemException& ex) {
        std::cerr << "A CORBA System Exception was caught: " << ex << std::endl;
    }

    // catch all CORBA non-system exceptions
    catch (CORBA::Exception& ex) {
        std::cerr << "A CORBA Exception was caught: " << ex << std::endl;
    }
}
```

In addition to printing specific fields of the exception with the _rep_id() and minor() member functions, it is possible to simply insert the entire exception into an output stream, such as std::cerr. For instance, the catch clauses of the above example can be rewritten as follows:

```c++
    catch (CORBA::SystemException& ex) {
        std::cerr << "A CORBA System Exception was caught: " << ex << std::endl;
    }

    // catch all CORBA non-system exceptions
    catch (CORBA::Exception& ex) {
        std::cerr << "A CORBA Exception was caught: " << ex << std::endl;
    }
```
In TAO, the output stream insertion operator \texttt{\small{\textless\textgreater}()} has been overloaded for CORBA exceptions to print some details of the exception, such as its unqualified type name and its interface repository id. Here is what is printed by the first catch clause above if a \texttt{CORBA:TRANSIENT} exception is raised:

A CORBA System Exception was caught: \texttt{TRANSIENT (IDL:omg.org/CORBA/TRANSIENT:1.0)}

**Example** Here is an example of a client performing the same remote invocations but catching specific system exceptions.

```c++
#include <corba.h>
#include <iostream>

int main(int argv, char* argc[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        Hotel_var hotel = // get a Hotel proxy, possibly through the Naming Service
        CORBA::String_var name = hotel->name();
        std::cout << "The name of the hotel is " << name << std::endl;

        CORBA::Short numNights = 5;
        GuestRoom_var room = hotel->checkIn(numNights);
        std::cout << "The room number is " << room->roomNumber() << std::endl;
        orb->destroy();
    } catch (CORBA::COMM_FAILURE& ex) {
        std::cerr << "A communication failure occurred: "
                   << "; minor code = " << ex.minor() << std::endl;
    }
    catch (CORBA::TRANSIENT& ex) {
        std::cerr << "A transient failure occurred: "
                   << "; minor code = " << ex.minor() << std::endl;
    }
    // catch all other system exceptions
    catch (CORBA::SystemException& ex) {
        std::cerr << "A CORBA System Exception was caught: ";
        std::cerr << "ID = " << ex._rep_id()
                   << "; minor code = " << ex.minor() << std::endl;
    }
    // catch all CORBA non-system exceptions
    catch (CORBA::Exception& ex) {
        std::cerr << "A CORBA Exception was caught: ";
        std::cerr << "ID = " << ex._rep_id() << std::endl;
    }
```
6.3 CORBA User Exceptions

Distributed system designers can use exceptions to communicate application-defined error conditions by specifying IDL modules and interfaces that contain CORBA User Exceptions. As shown in Figure 6-3, in C++, all user exceptions derive from `CORBA::UserException`, which itself derives from `CORBA::Exception`.

![Figure 6-3 User Exceptions](image)

A user exception is similar to an IDL `struct` in that it can contain data members. For example, you can extend the `GuestRoom` interface as follows:

```c++
interface GuestRoom
{
    exception Unoccupied
    {
        short daysEmpty;
        string lastOccupant;
    }

    readonly attribute short roomNumber;
    readonly attribute float balance;
}
```
void checkOut() raises (Unoccupied);
};

The client can catch the Unoccupied exception whenever GuestRoom::checkOut() is called:

GuestRoom_var room = // get a proxy to a GuestRoom;
try {
    room->checkOut();
}
catch (GuestRoom::Unoccupied& ex) {
    std::cerr << "Cannot check out. Room "
        << room->roomNumber() << " has been empty for "
        << ex.daysEmpty << " days. The last occupant was "
        << ex.lastOccupant << std::endl;
}
// catch all other user-defined CORBA exceptions
catch (CORBA::UserException& ex) {
    std::cerr << "A CORBA User Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id() << std::endl;
}
catch (CORBA::SystemException& ex) {
    std::cerr << "A CORBA System Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id()
        << "; minor code = " << ex.minor() << std::endl;
}
// should never be reached, because all CORBA exceptions are
// either system exceptions or user exceptions.
catch (CORBA::Exception& ex) {
    std::cerr << "A CORBA Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id() << std::endl;
}

On the server side, the server’s implementation of the checkOut() operation throws an Unoccupied exception if a checkOut() is attempted on an empty room. For example, suppose GuestRoom_i is the name of the server-side class that implements the GuestRoom interface:

void GuestRoom_i::checkOut()
{
    if ( /* the room is occupied */ ) {
        // check the guest out of the room
    }
    else { /* the room is unoccupied; ERROR */
        CORBA::Short daysEmpty = // # of days room is empty
        CORBA::String_var lastOccupant = // name of last occupant

        throw GuestRoom::Unoccupied (daysEmpty,lastOccupant);
    }
}
Conceptually, the exception is thrown “across the wire” to the client. Actually, what happens is a little more complicated. First, the exception thrown by the implementation is caught by the CORBA infrastructure (skeleton or ORB) on the server side, marshalled, and returned to the client in the reply message. The client-side CORBA infrastructure (ORB and stub) unmarshals the reply body into a C++ exception, then throws it. Finally, the client code can catch the exception and handle it in an application-specific manner. Conceptually, therefore, the exception appears to be thrown directly from the server to the client.

### 6.4 Error Handling Without Exception Handling

C++ exceptions provide a convenient way to communicate error conditions in a CORBA-based distributed system. Exceptions appear to be thrown directly from CORBA servers to clients, allowing system designers to concentrate on application and domain issues. However, C++ exceptions are not suitable in all situations. For example:

- Some C++ compilers and/or platforms do not support exceptions. For instance, older GNU C++ compilers and many real-time platforms have poor support for exceptions.
- Some real-time systems have such stringent performance and predictability requirements that they cannot risk the time or space overhead, nor the non-deterministic nature, of defining, throwing, or catching exceptions.
- Even in environments that do support C++ exceptions, it is possible to intentionally disable C++ exceptions when building TAO and distributed applications to minimize footprint and reduce the likelihood of subtle errors that arise from improper use of native C++ exceptions.

Here we describe error handling when exceptions are not supported by the compiler, or TAO has been configured to not use native C++ exceptions. See A.3 for details about building TAO without native exception handling.

When TAO is configured to not use native C++ exception handling, all system and user exceptions are returned via the `CORBA::Environment` parameter. Every method defined in user-defined interfaces will include a `CORBA::Environment` parameter as its last argument. This argument must be passed into user-defined remote operations as well as other CORBA/TAO
member functions that could report errors. It must be checked immediately afterward for the existence of any exceptions. The previous example’s client code now changes to:

```cpp
// Get a GuestRoom object reference (e.g., from the Hotel object).
GuestRoom_var room = ...;

// Invoke the checkOut() operation, passing a CORBA::Environment parameter.
CORBA::Environment env;
room->checkOut(env);

// Check to see if an exception occurred.
if (env.exception()) {
    // An exception occurred and was returned via env.
    // (The CORBA::Environment object keeps ownership of the exception.)
    CORBA::Exception* exc = env.exception();
    std::cout << exc->toString() << std::endl;
}
else {
    // It was not a GuestRoom::Unoccupied exception, so
    // deal with other types of exceptions here...
}
```

Additional blocks of code are added to the else clause to handle the other types of exceptions that were previously caught. On the server side, the implementation of `GuestRoom_i::checkOut()` changes to:

```cpp
void GuestRoom_i::checkOut(CORBA::Environment& env)
{
    if ( /* the room is occupied */ ) {
        // Check the guest out of the room.
    }
    else {
        // The room is unoccupied; ERROR!
        CORBA::Short daysEmpty = ...; // Number of days room is empty.
        CORBA::String_var lastOccupant = ...; // Name of last occupant.

        // Create a GuestRoom::Unoccupied exception and return it
        // via the CORBA::Environment parameter.
        GuestRoom::Unoccupied* unocc =
            new GuestRoom::Unoccupied(daysEmpty, lastOccupant);
        // (The CORBA::Environment object takes ownership of the exception.)
```
For more information on the alternate C++ mapping for systems lacking exception handling, see 1.42.2 of the OMG C++ Language Mapping Specification (OMG Document formal/03-06-03).

6.5 The ACE Exception Handling Macros

ACE provides macros that can help you write code that is portable between environments that support C++ exceptions and those that do not. In environments that support C++ exceptions, the ACE exception macros map to their corresponding C++ try and catch statements. In environments where C++ exceptions are not available, the macros map to C++ code constructs that simulate native C++ exceptions using the CORBA::Environment parameter. The behavior of these macros is controlled by the ACE_HAS_EXCEPTIONS macro that is typically set automatically based on the setting of the exceptions flag in the platform_macros.GNU file or the MAKEFLAGS environment variable. See A.3 for details on setting the exceptions flag when building ACE and TAO.

The ACE exception macros, described in the following paragraphs, can only be used with CORBA and cannot be used to simulate general C++ exceptions. Each macro name starts with the prefix ACE_ and can be easily ported to ORBs other than TAO. The macros depend on the CORBA::Environment parameter that communicates exception information in systems using the alternate C++ mapping for exception handling.

Note

The ACE exception macros used by TAO were changed shortly before the release of TAO 1.2a. New macros were introduced that expand to nothing when native C++ exceptions are in use so that extra CORBA::Environment parameters no longer need to be propagated throughout TAO’s internal and your application’s APIs. If you have existing application code that uses the older macros, you can achieve backward compatibility by using the include_env=1 GNU Make flag or by inserting #define ACE_ENV_BKWD_COMPAT in your config.h file. These backward compatibility settings are only applicable when native C++ exceptions are
enabled (e.g., via exceptions=1). See Appendix A for more information on setting these options.

**ACE_DECLARE_NEW_ENV**
Declarations a new CORBA::Environment object called ACE_TRY_ENV. All of the other macros use this CORBA::Environment object.

**ACE_TRY and ACE_TRY_EX**
Simulates a C++ try. Use ACE_TRY to simulate a single try block. Use ACE_TRY_EX when more than one try block is needed in the same function body. The parameter given to ACE_TRY_EX is used as a label that the macro uses to control the flow of the program in the event of an exception.

**ACE_TRY_NEW_ENV**
Combines the functionality of the macros ACE_TRY and ACE_DECLARE_NEW_ENV.

**ACE_ENV_ARG_DECL**
Used when declaring a method within a class or defining a method implementation with at least one parameter. There can be no comma between the last parameter and this macro. When native exceptions are enabled, this macro is empty. Otherwise, it defines a CORBA::Environment parameter named ACE_TRY_ENV.

**ACE_ENV_ARG_DECL_WITH_DEFAULTS**
Used when declaring a method within a class with at least one parameter. There can be no comma between the last parameter and this macro. It performs the same function as the ACE_ENV_ARGDECL macro, but provides a default value.

**ACE_ENV_ARG_DECL_NOT_USED**
Used when defining a method implementation with at least one parameter. There can be no comma between the last parameter and this macro. When native exceptions are enabled, this macro is empty. Otherwise, it defines an unnamed CORBA::Environment parameter.
ACE_ENV_SINGLE_ARG_DECL
Used when declaring a method within a class or defining a method implementation with no parameters. When native exceptions are enabled, this macro is empty. Otherwise, it defines a CORBA::Environment parameter named ACE_TRY_ENV.

ACE_ENV_SINGLE_ARG_DECL_WITH_DEFAULTS
Used when declaring a method within a class with no parameters. It performs the same function as the ACE_ENV_SINGLE_ARG_DECL macro, but provides a default value.

ACE_ENV_SINGLE_ARG_DECL_NOT_USED
Used when defining a method implementation with no parameters. When native exceptions are enabled, this macro is empty. Otherwise, it defines an unnamed CORBA::Environment parameter.

ACE_ENV_ARG_PARAMETER
This macro must be passed into each remote operation that takes at least one parameter. There can be no comma between the last parameter and this macro. When native exceptions are enabled, this macro is empty. Otherwise, the ACE_TRY_ENV CORBA::Environment object is passed.

ACE_ENV_SINGLE_ARG_PARAMETER
This macro must be passed into each remote operation that takes no parameters.

ACE_ENDTRY
Used after the last ACE_CATCH of an ACE_TRY/ACE_CATCH block.

ACE_THROW and ACE_THROW_RETURN
Throw an exception from outside an ACE_TRY block.

ACE_TRY_THROW and ACE_TRY_THROW_EX
Throw an exception from inside an ACE_TRY block. Use ACE_TRY_THROW_EX in conjunction with an ACE_TRY_EX. The parameters to ACE_TRY_THROW_EX are the exception type and the label used in the corresponding ACE_TRY_EX.
**ACE_RE_THROW and ACE_RE_THROW_EX**
Propagate an exception from inside an ACE_CATCH. Use ACE_RE_THROW_EX in conjunction with an ACE_TRY_EX. The parameter to ACE_RE_THROW_EX is the label used in the corresponding ACE_TRY_EX.

**ACE_CATCH**
Simulates a C++ catch. The parameters to this macro are the exception type and exception variable name.

**ACE_CATCHANY**
Simulates C++ catch CORBA::Exception& ACE_ANY_EXCEPTION. The exception is stored in the variable ACE_ANY_EXCEPTION.

**ACE_CATCHALL**
Simulates a C++ catch (...).

**ACE_TRY_CHECK and ACE_TRY_CHECK_EX**
Check for an exception from inside an ACE_TRY or ACE_TRY_EX block. Checks the ACE_TRY_ENV CORBA::Environment object to see if an exception has been thrown; if so, execution of the program is redirected to the ACE_CATCH clauses to simulate the behavior of a native C++ exception. Insert this macro after each remote invocation that is inside an ACE_TRY or ACE_TRY_EX block. The ACE_TRY_CHECK_EX macro parameter is the label used in the corresponding ACE_TRY_EX.

**ACE_CHECK and ACE_CHECK_RETURN**
Check for an exception from outside an ACE_TRY block. Checks the CORBA::Environment object ACE_TRY_ENV to see if an exception has been thrown; if so, exception propagation is simulated by returning from the current function. To return a value from a non-void function, use ACE_CHECK_RETURN.

**ACE_PRINT_EXCEPTION**
Prints an exception. The macro takes two arguments: the exception itself; and a string containing extra information. This macro is useful for debugging as it prints TAO-specific information interpreted from the CORBA::SystemException’s minor code, including information about the
source code location (in the ORB) of the origin of the exception and error number decoding.

**ACE_THROW_SPEC**

This macro is used with method definitions to indicate that a method can throw an exception. It is defined to nothing if the macro ACE_HAS_NO_THROW_SPEC is defined or exceptions are disabled. Otherwise, it is the same as a C++ throw.

As you can see, there are several versions of TRY, several versions of CATCH, and several versions of THROW, as well as several versions of the checking macros. This complexity is necessary to allow identical code to be written for environments with and without support for C++ exceptions. In 6.5.1 through 6.5.5, we present examples showing how to use ACE exception macros.

### 6.5.1 ACE_TRY/ACE_CATCH Block

This example duplicates the functionality of the client code in 6.2. Differences between this example and the first example presented in 6.2 appear in **bold** type.

```cpp
#include <corba.h>
#include <iostream>

int main(int argv, char* argc[]) {
    ACE_TRY_NEW_ENV {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "" ACE_ENV_ARG_PARAMETER);
        ACE_TRY_CHECK;

        Hotel_var hotel = // get a Hotel proxy, possibly through the Naming Service
        ACE_TRY_CHECK;

        CORBA::String_var name = hotel->name(ACE_ENV_SINGLE_ARG_PARAMETER);
        ACE_TRY_CHECK;
        std::cout << "The name of the hotel is " << name << std::endl;

        CORBA::Short numNights = 5;
        GuestRoom_var room = hotel->checkIn(numNights ACE_ENV_ARG_PARAMETER);
        ACE_TRY_CHECK;

        CORBA::Short roomNumber = room->roomNumber(ACE_ENV_SINGLE_ARG_PARAMETER);
        ACETRY_CHECK;
        std::cout << "The room number is " << roomNumber << std::endl;
```
// Release resources.
orb->destroy(ACE_ENV_SINGLE_ARG_PARAMETER);
ACE_TRY_CHECK;
}
ACE_CATCH (CORBA::COMM_FAILURE, ex)
{
  ACE_PRINT_EXCEPTION(ex, "A communication failure occurred.");
}
ACE_CATCH (CORBA::NO_PERMISSION, ex) {
  ACE_PRINT_EXCEPTION(ex, "A permission failure occurred.");
}
// catch all other system exceptions
ACE_CATCH (CORBA::SystemException, ex)
{
  ACE_PRINT_EXCEPTION(ex, "A CORBA System Exception occurred.");
}
// catch all CORBA non-system exceptions
ACE_CATCHANY
{
  ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION, "A CORBA Exception occurred.");
}
ACE_CATCHALL
{
  std::cerr << "An unknown exception was caught" << std::endl;
}
ACE_ENDTRY;

ACE_CHECK_RETURN(-1);
return 0;

In an environment that has C++ exceptions, both of these examples function in exactly the same way. In an environment that does not have C++ exceptions, the ACE_TRY macros simulate the control flow of exceptions.

The differences are as follows:

- **ACE_TRY_NEW_ENV** creates a new CORBA::Environment object called ACE_TRY_ENV that is used by the ORB to communicate error information back to the calling function. The name ACE_TRY_ENV is very important because many other macros use it implicitly.

  Normally, we only need to create a new CORBA::Environment if an environment object is not already available. For example, when a function that receives a CORBA::Environment object as a parameter calls a function requiring a CORBA::Environment parameter, the environment
object received by the first function should be passed on to the second. It is not necessary to create a new CORBA::Environment variable in this situation.

ACE_TRY_NEW_ENV then simulates the C++ keyword try.

- The CORBA::ORB_init() call can potentially throw an exception. The ACE_ENV_ARG_PARAMETER macro must be passed to CORBA::ORB_init() as the last parameter to ensure that error information is communicated, even if the compiler/platform does not have C++ exceptions.

If CORBA::ORB_init() throws an exception, then the ACE_TRY_ENV object will contain exception information. The ACE_TRY_CHECK macro checks for an exception. If one is found, control passes to the ACE_CATCH clauses. The ACE_TRY_ENV object simulates the behavior of native C++ exceptions. If an exception is thrown, control passes directly to the catch clauses. There should be an ACE_CHECK or an ACE_TRY_CHECK after every invocation to which an ACE_ENV_ARG_PARAMETER or ACE_ENV_SINGLE_ARG_PARAMETER is passed.

- The code that gets the Hotel’s object reference is not shown, but it probably involves the CORBA Naming Service or some other remote object. Thus, an ACE_TRY_CHECK is necessary to check for exceptions and re-route control appropriately.

- The operation hotel->name() is a remote operation, so you must pass an ACE_ENV_SINGLE_ARG_PARAMETER as its only parameter and check it with ACE_TRY_CHECK after the call completes.

- The operation hotel->checkIn() is also a remote operation, so you must pass ACE_ENV_ARG_PARAMETER and check for exceptions with ACE_TRY_CHECK when it completes.

- The operation room->roomNumber() is also a remote operation, but here you must be careful not to call it in the middle of an iostream operation. If an error occurs during the room->roomNumber() operation, then erroneous data writes to standard output before you have a chance to check for exceptions. To avoid this, separate the room->roomNumber() remote operation from the output statement. In general, avoid calling remote operations in the middle of statements that have other side effects.
6.5 The ACE Exception Handling Macros

- The first three ACE_CATCH clauses are similar to their C++ catch counterparts. The ACE_CATCH macro takes the following two parameters:
  - The C++ class of the exception being caught.
  - A variable name for a CORBA::Exception object that is to be populated with the exception information.

The ACE_CATCH macro always catches exceptions by reference. These three examples catch CORBA system exceptions, but CORBA user exceptions are caught in exactly the same way.

Note In the example, catch clauses use ACE_PRINT_EXCEPTION to print exception contents. This macro takes the exception itself and a string containing extra information as its two arguments.

- The ACE_CATCHANY macro replaces catch (CORBA::Exception& ex). Do not confuse this with ACE_CATCHALL. If you use ACE_CATCHANY, it must be the last catch clause that catches CORBA exceptions.
- The ACE_CATCHALL macro replaces catch (...). As is the case with catch(...), use ACE_CATCHALL infrequently.
- ACE_ENDTRY always follows the last catch clause of a try/catch block. It is very important to end every try/catch block with ACE_ENDTRY.
- It is a good idea to follow each ACE_ENDTRY with either an ACE_CHECK (if outside an ACE_TRY/ACE_CATCH block) or ACE_TRY_CHECK (if inside an ACE_TRY/ACE_CATCH block). There may be active exceptions that have not been caught by any of the ACE_CATCH clauses, and you do not want to ignore them.

6.5.2 Throwing Exceptions with ACE_THROW

There are four situations in which an exception can be thrown:
- An exception can be thrown from outside an ACE_TRY block. For this situation, use ACE_THROW. This macro’s parameter is a constructor for an exception object. Do not pass the exception object itself.

```c
void GuestRoom_i::checkOut(ACE_ENV_SINGLE_ARG_DECL)
{
    if (/* the room is occupied */)
    // check the guest out of the room
```
Error Handling

void GuestRoom_i::checkOut(ACE_ENV_SINGLE_ARG_DECL)
{
    ACE_TRY
    {
        if ( /* the room is occupied */ ) {
            // check the guest out of the room
        }
    } else { // the room is unoccupied; ERROR
        CORBA::Short daysEmpty = // # of days room is empty
        CORBA::String_var lastOccupant = // name of last occupant

        ACE_THROW(GuestRoom::Unoccupied (daysEmpty,lastOccupant));
    }
}

• An exception can be thrown from inside an ACE_TRY block or an
ACE_CATCH clause. For this situation, use ACE_TRY_THROW. The macro’s
parameter is a constructor for an exception object. Do not pass the
exception object itself.

void GuestRoom_i::checkOut(ACE_ENV_SINGLE_ARG_DECL)
{
    ACE_TRY
    {
        if ( /* the room is occupied */ ) {
            // check the guest out of the room
        }
    } else { // the room is unoccupied; ERROR
        CORBA::Short daysEmpty = // # of days room is empty
        CORBA::String_var lastOccupant = // name of last occupant

        ACE_TRY_THROW(GuestRoom::Unoccupied (daysEmpty,lastOccupant));
    }
}

ACE_CATCH(CORBA::SystemException& ex)
{
    const char* str = "CORBA System Exception in GuestRoom_i::checkOut.";
    ACE_PRINT_EXCEPTION(ex, str);

    ACE_TRY_THROW(xmsg(str));
}

• When you need more than one simulated try/catch block within a
given scope, use ACE_TRY_EX rather than ACE_TRY to distinguish the
individual blocks (See 6.5.4.). An exception can be thrown from inside an
ACE_TRY_EX block. For this situation, use ACE_TRY_THROW_EX. The
macro’s parameters are a constructor for an exception object and the name
of the enclosing ACE_TRY_EX block.

void GuestRoom_i::checkOut(ACE_ENV_SINGLE_ARG_DECL)
{ 
    ACE_TRY_EX(CheckOutBlock)
}
6.5 The ACE Exception Handling Macros

```c
{    
if ( /* the room is occupied */ ) {
    // check the guest out of the room
} else { // the room is unoccupied; ERROR
    CORBA::Short daysEmpty =         // # of days room is empty
    CORBA::String_var lastOccupant = // name of last occupant

    ACE_TRY_THROW_EX(GuestRoom::Unoccupied (daysEmpty,lastOccupant),
                     CheckOutBlock);
}

ACE_CATCH(CORBA::SystemException& ex)
{    
    const char* str = "CORBA System Exception in GuestRoom_i::checkOut.";
    ACE_PRINT_EXCEPTION(ex, str);
    ACE_TRY_THROW_EX(xmsg(str),CheckOutBlock);
}

ACE_ENDTRY;
}

• An exception can be propagated from inside an ACE_CATCH clause. Use
ACE_RE_THROW to propagate an exception from an ACE_CATCH clause
that is part of an ACE_TRY block. Use ACE_RE_THROW_EX to propagate an
exception from an ACE_CATCH clause that is part of an ACE_TRY_EX
block. For example:

void GuestRoom_i::checkOut(ACE_ENV_SINGLE_ARG_DECL)
{
    ACE_TRY
    {
    if ( /* the room is occupied */ ) {
    // check the guest out of the room
    } else { // the room is unoccupied; ERROR
    CORBA::Short daysEmpty =         // # of days room is empty
    CORBA::String_var lastOccupant = // name of last occupant

    ACE_TRY_THROW(GuestRoom::Unoccupied ex(daysEmpty,lastOccupant));
    }
    }
ACE_CATCH(CORBA::SystemException& ex)
{    
    const char* str = "CORBA System Exception in GuestRoom_i::checkOut.";
    ACE_PRINT_EXCEPTION(ex, str);
    ACE_RE_THROW;
}
ACE_ENDTRY;
}
```
Error Handling

**Note**  
The `ACE_THROW`, `ACE_TRY_THROW`, and `ACE_TRY_THROW_EX` macros require a constructor, not an object, as the exception parameter. If an object is passed, the code does not compile.

The following code compiles properly:

```cpp
CORBA::Short daysEmpty = // # of days room is empty
CORBA::String_var lastOccupant = // name of last occupant

ACE_THROW(GuestRoom::Unoccupied (daysEmpty,lastOccupant));
```

The following code does not compile:

```cpp
CORBA::Short daysEmpty = // # of days room is empty
CORBA::String_var lastOccupant = // name of last occupant

// Do not do this!!!
GuestRoom::Unoccupied ex(daysEmpty,lastOccupant);
ACE_THROW(ex);
```

6.5.3 Returning Values When Simulating Exceptions

The `ACE_CHECK_RETURN` and `ACE_THROW_RETURN` macros are special versions of the `ACE_CHECK` and `ACE_THROW` macros. In environments that have C++ exceptions, `ACE_CHECK` and `ACE_CHECK_RETURN` are identical, as are `ACE_THROW` and `ACE_THROW_RETURN`. In environments that do not have native C++ exceptions, however, these macros give us the ability to write code that compiles successfully and functions properly.

In an environment with native C++ exceptions, a function throwing an exception does not return a value. In other words, each function call either returns a value (or has a `void` return type) or throws an exception, but never both.

This is not the case in an environment without native C++ exceptions. Each function call either returns a value of some kind or returns `void`; no real exceptions are thrown. The exception information is passed back in the `ACE_TRY_ENV` object.

On compilers/platforms that do not have C++ exceptions, the `ACE_CHECK` and `ACE_THROW` macros both contain `return` statements for exiting from their enclosing functions. In this example, `ACE_THROW` calls `return` to exit from the `GuestRoom_i::checkOut()` function:
void GuestRoom_i::checkOut(ACE_ENV_SINGLE_ARG_DECL)
{
    if ( /* the room is occupied */ ) {
        // check the guest out of the room
    }
    else { // the room is unoccupied; ERROR
        CORBA::Short daysEmpty =          // # of days room is empty
        CORBA::String_var lastOccupant = // name of last occupant

        ACE_THROW(GuestRoom::Unoccupied (daysEmpty,lastOccupant));
    }
}

However, if we change the signature of the checkOut() function so that it returns a CORBA::Short value instead of a void, then ACE_CHECK and ACE_THROW(ex) may cause compilation errors or warnings. We need the following:

- A mechanism to check for an exception and return a value simultaneously.
- A mechanism to “throw” an exception and return a value simultaneously.

We use ACE_CHECK_RETURN and ACE_THROW_RETURN to accomplish this:

CORBA::Short GuestRoom_i::checkOut(ACE_ENV_SINGLE_ARGDECL)
{
    if ( /* the room is occupied */ ) {
        // check the guest out of the room

        CORBA::Short roomNum = this->roomNumber(ACE_ENV_SINGLE_ARG_PARAMETER);
        ACE_CHECK_RETURN(-1);
        return roomNum;
    }
    else { // the room is unoccupied; ERROR
        CORBA::Short daysEmpty =          // # of days room is empty
        CORBA::String_var lastOccupant = // name of last occupant

        ACE_THROW_RETURN(GuestRoom::Unoccupied (daysEmpty,lastOccupant), -1);
    }
}

Note  We have chosen -1 as our “bogus” room number value. In any situation where an exception is thrown, clients that catch exceptions should not depend on the return value being anything in particular.
In the following example, the `main()` function returns `-1` if an exception is thrown and `0` otherwise.

```cpp
#include <corba.h>
#include <iostream>

int main(int argv, char* argc[]) { 
    ACE_DECLARE_NEW_ENV;
    ACE_TRY
    { 
        // ...Initialize the ORB, get the Hotel proxy...
        CORBA::String_var name = hotel->name(ACE_ENV_SINGLE_ARG_PARAMETER);
        ACE_TRY_CHECK;
        std::cout << "The name of the hotel is " << name << std::endl;
        orb->destroy(ACE_ENV_SINGLE_ARG_PARAMETER);
        ACE_TRY_CHECK;
    } // catch all CORBA exceptions
    ACE_CATCHANY
    { 
        ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION, "A CORBA Exception occurred.");
    }
    ACE_CATCHALL
    { 
        std::cerr << "An unknown exception was caught" << std::endl;
    }
    ACE_ENDTRY;
    ACE_CHECK_RETURN(-1);
    return 0;
}
```

### 6.5.4 Multiple ACE_TRY/ACE_CATCH Blocks

In C++, it is possible to have more than one `try/catch` block within the same block of code. For example, you could re-write the client code as follows:

```cpp
#include <corba.h>
#include <iostream>

int main(int argv, char* argc[]) { 
    // ... ORB_init, etc ...
```
Hotel_var hotel;
try {
    hotel = // get a Hotel proxy, possibly through the Naming Service
    CORBA::String_var name = hotel->name;
    std::cout << "The name of the hotel is " << name << std::endl;
} catch (CORBA::SystemException & ex) {
    std::cerr << "A CORBA System Exception was caught." << std::endl;
    return -1;
}

try {
    CORBA::Short numNights = 5;
    GuestRoom_var room = hotel->checkIn(numNights);
    std::cout << "The room number is " << room->roomNumber() << std::endl;
} catch (CORBA::SystemException & ex) {
    std::cerr << "A CORBA System Exception was caught." << std::endl;
    return -1;
}

return 0;

You can re-write this code using the ACE_TRY macros, but you must use caution. The ACE_TRY macro declares a label, and that label always has the name ACE_TRY_LABEL. If two ACE_TRY declarations are encountered in the same code block, a compiler error results. The ACE_TRY_EX macro solves this problem.

The ACE_TRY_EX macro has one parameter, and the ACE_TRY*EX macros use that parameter internally as a label. For example:

#include <corba.h>
#include <iostream>

int main(int argc, char* argv[]) {

    // ... ORB_init, etc ...

    Hotel_var hotel;
    ACE_DECLARE_NEW_ENV;
    ACE_TRY_EX(Block_1)
    {
        hotel = // get a Hotel proxy, possibly through the Naming Service
        ACE_TRY_CHECK_EX(Block1);

        CORBA::String_var name = hotel->name(ACE_ENV_SINGLE_ARG_PARAMETER);
        ACE_TRY_CHECK_EX(Block1);
    }

    // ...
std::cout << "The name of the hotel is " << name << std::endl;
)
ACE_CATCH (CORBA::SystemException, ex)
{
    ACE_PRINT_EXCEPTION(ex, "A CORBA System Exception was caught.");
    return -1;
}
ACE_ENDTRY;
ACE_CHECK_RETURN(-1);

ACE_TRY_EX(Block_2)
{
    CORBA::Short numNights = 5;
    GuestRoom_var room = hotel->checkIn(numNights ACE_ENV_ARG_PARAMETER);
    ACE_TRY_CHECK_EX(Block2);
    CORBA::Short roomNumber = room->roomNumber(ACE_ENV_SINGLE_ARG_PARAMETER);
    ACE_TRY_CHECK_EX(Block2);
    std::cout << "The room number is " << roomNumber << std::endl;
}
ACE_CATCH (CORBA::SystemException, ex)
{
    ACE_PRINT_EXCEPTION(ex, "A CORBA System Exception was caught.");
    return -1;
}
ACE_ENDTRY;
ACE_CHECK_RETURN(-1);
return 0;
}

In the above code, notice the use of ACE_TRY_EX instead of ACE_TRY and the use of ACE_TRY_CHECK_EX instead of ACE_TRY_CHECK. The *_EX macros differentiate between code blocks so that the ACE_TRY_CHECK_EX code knows which ACE_TRY_EX block it belongs to. The example is otherwise unchanged.

### 6.5.5 Nested ACE_TRY/ACE_CATCH blocks

In rare cases you may nest C++ try/catch blocks. The following example nests ACE_TRY/ACE_CATCH blocks by using the macros ACE_TRY_EX and ACE_TRY_CHECK_EX:

```cpp
int main(int argv, char* argc[])
{
    // ... ORB_init, etc ...

    Hotel_var hotel;
    ACE_DECLARE_NEW_ENV;
    ACE_TRY_EX(OuterBlock)
    {
        hotel = // get a Hotel proxy, possibly through the Naming Service
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
6.6 Synopsis of TAO’s Error Handling Guidelines

The following items provide a synopsis of TAO’s error handling guidelines:

- Always end an `ACE_TRY` block with `ACE_ENDTRY`.
- Always place an `ACE_TRY_CHECK` after each remote operation. This is a very important step in simulating C++ exceptions in an exception-less environment.
Error Handling

The following example illustrates the danger of not using an ACE_TRY_CHECK after each remote operation:

```cpp
ACE_DECLARE_NEW_ENV;
// assume that "hotel" represents a valid Hotel object reference
ACE_TRY
{
    CORBA::Short numNights = 5;
    CORBA::GuestRoom_var room = hotel->checkIn(numNights
        ACE_ENV_ARG_PARAMETER);

    CORBA::Short roomNumber =
        room->roomNumber(ACE_ENV_SINGLE_ARG_PARAMETER);
    std::cout << "Successfully checked into room " << roomNumber <<
        std::endl;
}
ACE_CATCH (CORBA::SystemException, ex)
{
    ACE_PRINT_EXCEPTION(ex, "A CORBA System Exception was caught.");
    return -1;
}
// catch all CORBA non-system exceptions
ACE_CATCHANY
{
    ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION, "A CORBA Exception occurred.");
}
ACE_ENDTRY;
ACE_CHECK;
```

The above code behaves differently depending on whether or not the compiler/platform supports C++ exceptions. If the environment supports exceptions, and hotel->checkIn() throws an exception, then control immediately passes to the ACE_CATCH blocks, and the "Successfully checked into room" message is not printed. If the environment does not support exceptions, then the failure of hotel->checkIn() goes unnoticed, and the message "Successfully checked into room" is printed. In fact, since the GuestRoom proxy is invalid, another exception is thrown during the call to room->roomNumber(), but that exception is ignored also.

If you make the following changes to the example, then the behavior is exactly the same, whether native C++ exceptions are used or not:

```cpp
ACE_DECLARE_NEW_ENV;
// assume that "hotel" is a pointer to a valid Hotel proxy
ACE_TRY
```
6.6 Synopsis of TAO's Error Handling Guidelines

```cpp
{ 
  CORBA::Short numNights = 5;
  CORBA::GuestRoom_var room = hotel->checkIn(numNights
    ACE_ENV_ARG_PARAMETER);

  ACE_TRY_CHECK;

  CORBA::Short roomNumber =
    room->roomNumber(ACE_ENV_SINGLE_ARG_PARAMETER);

  ACE_TRY_CHECK;
  std::cout << "Successfully checked into room " << roomNumber <<
            std::endl;
}
ACE_CATCH (CORBA::SystemException, ex)
{
  ACE_PRINT_EXCEPTION(ex, "A CORBA System Exception was caught.");
  return -1;
}
// catch all CORBA non-system exceptions
ACE_CATCHANY
{
  ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION,"A CORBA Exception occurred.");
}
ACE_ENDTRY;
ACE_CHECK;

In environments where exceptions are supported, this example behaves
the same as before. However, in environments where exceptions are not
supported, an ACETRYCHECK throws control to the ACE_CATCH clauses
when an exception has occurred, eliminating erroneous messages.

- Do not call a remote operation within a statement that has side effects. The
code behaves differently in exception-supporting environments than it
does in non-exception-supporting environments. For example:

ACE_DECLARE_NEW_ENV;
// assume that "hotel" is a pointer to a valid Hotel proxy
ACE_TRY
{
  CORBA::Short numNights = 5;
  CORBA::GuestRoom_var room = hotel->checkIn(numNights
    ACE_ENV_ARG_PARAMETER);

  ACE_TRY_CHECK;

  std::cout << "Successfully checked into room "
            << room->roomNumber(ACE_ENV_SINGLE_ARG_PARAMETER)
            << std::endl;
  ACE_TRY_CHECK;
}
The above example appears to make proper use of the `ACE_TRY_CHECK` macro to check for an exception after each remote operation. However, suppose the `room->roomNumber()` operation throws an exception. In environments that have native C++ exceptions, the "Successfully checked into room" message does not print. In environments that do not have native C++ exceptions, the message prints, probably with a bogus room number. This is an undesirable side effect.

The side effect may be worse if it results in a counter incrementing erroneously or an attempt to manipulate a NULL pointer.

- **Avoid using break and continue statements in ACE_TRY blocks.** The `ACE_TRY` macro code is actually a do/while loop, so break and continue statements cause the code to jump to unexpected places. In a non-exception environment, the following code functions improperly:

```cpp
ACE_DECLARE_NEW_ENV;
for (int i = 0; i < 100; ++i) {
  ACE_TRY
  {
    if (i > 90) {
      continue;     // will be confused by ACE_TRY
    }
    else if (i > 50) {
      break;        // will be confused by ACE_TRY
    }
    CORBA::Short numNights = 5;
    CORBA::GuestRoom_var room = hotel->checkIn(numNights
        ACE_ENV_ARG_PARAMETER);
  }
  ACE_TRY_CHECK;
}
CORBA::Short roomNumber =
  room->roomNumber(ACE_ENV_SINGLE_ARG_PARAMETER);
```
ACE_TRY_CHECK;
std::cout << "Successfully checked into room " << roomNumber << std::endl;
}
ACE_CATCH (CORBA::SystemException, ex)
{
    ACE_PRINT_EXCEPTION(ex, "A CORBA System Exception was caught.");
    return -1;
}
// catch all CORBA non-system exceptions
ACE_CATCHANY
{
    ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION,
        "A CORBA Exception occurred.");
}
ACE_ENDTTRY;
ACE_CHECK;
}

- Use ACE_TRY_CHECK and ACE_TRY_THROW with ACE_TRY.
- Use ACE_TRY_CHECK_EX and ACE_TRY_THROW_EX with ACE_TRY_EX.
- Always use ACE_TRY_* inside an ACE_TRY block.

6.7 TAO Minor Codes

A CORBA system exception includes information about the problem that occurred. You can obtain this information by calling CORBA::Exception::minor(). The value returned from minor() is a 32-bit value known as the minor code value. The minor code value contains several pieces of information about the exception in the following groups of bits:

- The high order 20 bits of the minor code value define the particular ORB implementation that is transmitting the exception. The OMG assigns this value to the ORB vendor or other responsible party to ensure it is unique for all ORB implementations. TAO’s unique identifier is the hexadecimal value 0x54410 (represented by ASCII "TA" followed by 0x0).

- The low order 12 bits are assigned no special significance by the OMG. These bits comprise the implementation-specific minor code. For TAO, the low order 12 bits are further divided as follows:
- The first 5 bits comprise the **location code** that identifies the location in TAO where the exception was raised.
- The remaining 7 bits encode an error number (**errno**) associated with the exception.

TAO’s location codes are described in 6.7.1 and the meanings of various error numbers are described in 6.7.2.

### 6.7.1 Location Codes

Each location code is assigned a preprocessor macro definition for improved source code readability. The file `$TAO_ROOT/tao/corbafwd.h` contains the definitions of these macros. To determine why an exception is being thrown, you can step through the TAO source code in a debugger. However, that can be time consuming. Instead, you can search the TAO source code for these macro definitions and examine the code that detected the failure. Doing so often leads to insights as to the exact cause of the error.

**0x01U TAO_INVOCATION_LOCATION_FORWARD_MINOR_CODE**

If a client attempts to connect to a server and receives a `LOCATION_FORWARD` reply, it retries the connection at the address contained in the reply. If the retry fails, the client ORB raises a `CORBA::TRANSIENT` exception with the minor code set to `TAO_INVOCATION_LOCATION_FORWARD_MINOR_CODE`.

**0x02U TAO_INVOCATION_SEND_REQUEST_MINOR_CODE**

If a client attempts to send a request to a server for the current profile and fails, the client ORB raises a `CORBA::TRANSIENT` exception with the minor code set to `TAO_INVOCATION_SEND_REQUEST_MINOR_CODE`.

**0x03U TAO_POA_DISCARDING**

A POAManager in the `PortableServer::POAManager::DISCARDING` state causes the associated POAs to discard all incoming requests. Requests for which processing has already begun are allowed to continue. When a request is discarded, the POA will raise a `CORBA::TRANSIENT` exception to indicate that the client should retry the request. The POA will set the minor code to `TAO_POA_DISCARDING`.

**0x04U TAO_POA_HOLDING**

A POAManager in the `PortableServer::POAManager::HOLDING` state may cause the POAs associated with it to queue incoming requests, up to an implementation-defined limit. If this limit is exceeded, the POA may discard requests and raise the `CORBA::TRANSIENT` exception with minor code of `TAO_POA_HOLDING` to indicate to the client that it should retry the request.
**Note**  
TAO’s implementation of the POA does not support queuing of requests, so requests are immediately rejected with a `CORBA::TRANSIENT` exception if the POAManager is in the HOLDING state.

0x05U TAO_UNHANDLED_SERVER_CXX_EXCEPTION  
If a servant implementation throws a native C++ exception that is not handled within the application code, the server ORB cannot propagate the exception to the client. Because the CORBA specification provides no standards for marshaling or demarshaling native C++ exceptions, the server ORB raises a `CORBA::UNKNOWN` exception with the minor code set to TAO_UNHANDLED_SERVER_CXX_EXCEPTION.

0x06U TAO_INVOCATION_RECV_REQUEST_MINOR_CODE  
If a client sends a request to a server and detects an error (other than a timeout) while waiting for a reply, the client ORB raises a `CORBA::COMM_FAILURE` exception with a TAO_INVOCATION_RECV_REQUEST_MINOR_CODE minor code.

0x07U TAO_CONNECTOR_REGISTRY_NO_USABLE_PROTOCOL  
If none of the connector objects in the ORB’s Connector_Registry are able to parse a particular URL-style stringified IOR (e.g., it may not be formatted properly or it may specify an unrecognized protocol), the ORB raises a `CORBA::INV_OBJREF` exception with the minor code set to TAO_CONNECTOR_REGISTRY_NO_USABLE_PROTOCOL. See Chapter 15 for more information on using TAO’s pluggable protocols and 19.8.10 for more information on specifying URL-style object references.

0x08U TAO_MPROFILE_CREATION_ERROR  
If the ORB, in attempting to parse a string (e.g., a stringified IOR), encounters an error creating the MProfile (the list of profiles contained within the object reference), it raises a `CORBA::INV_OBJREF` exception with the minor code set to TAO_MPROFILE_CREATION_ERROR.

0x09U TAO_TIMEOUT_CONNECT_MINOR_CODE  
If a client fails to connect to a server within a specified timeout period, the client ORB raises a `CORBA::TIMEOUT` exception with the minor code set to TAO_TIMEOUT_CONNECT_MINOR_CODE.

0x0AU TAO_TIMEOUT_SEND_MINOR_CODE  
If a client attempts to invoke an operation on a CORBA object, but the invocation can not be completed within a specified timeout period, the client
ORB raises a CORBA::TIMEOUT exception with the minor code set to TAO_TIMEOUT_SEND_MINOR_CODE.

0x0BU TAO_TIMEOUT_RECV_MINOR_CODE
If a client sends a request to a server but fails to receive a reply within a specified timeout period, the client ORB raises a CORBA::TIMEOUT exception with the minor code set to TAO_TIMEOUT_RECV_MINOR_CODE.

0x0CU TAO_IMPLREPO_MINOR_CODE
General-purpose code indicating an error related to the TAO Implementation Repository (IMR). For example, a server could not notify the IMR of its start up, the IMR could not forward a client request to a server, or a server is not running and the server’s activation mode does not allow the IMR to automatically start it. In all cases, a CORBA::TRANSIENT exception is raised with the minor code set to TAO_IMPLREPO_MINOR_CODE.

0x0DU TAO_ACCEPTOR_REGISTRY_OPEN_LOCATION_CODE
A server ORB opens an acceptor to allow it to accept client connections on one or more endpoints. Endpoints can be specified when the ORB is initialized (e.g., via the -ORBListenEndpoints option). If no endpoints are specified, the ORB will attempt to open acceptors on default endpoints for the loaded transports protocols. Opening an acceptor can fail for a variety of reasons, including: The ORB is unable to create or open an acceptor, the ORB is unable to add an acceptor to its acceptor registry, an invalid endpoint was specified, no usable transport protocol has been loaded, or a specified endpoint is already in use by another service. Such errors usually result in a CORBA::BAD_PARAM exception being raised with the minor code set to TAO_ACCEPTOR_REGISTRY_OPEN_LOCATION_CODE, usually during POA activation.

0x0EU TAO_ORB_CORE_INIT_LOCATION_CODE
An error can occur during ORB core initialization for a variety of reasons, including: An invalid argument was supplied to an -ORBInitRef option, an unrecognized argument starting with "-ORB" was passed to CORBA::ORB_init(), or an invalid endpoint was specified. In these cases, the ORB will raise CORBA::BAD_PARAM. Other reasons ORB initialization can fail include internal errors in TAO’s configuration causing the ORB to fail to load a resource factory or server strategy factory. In these cases, the ORB will raise CORBA::INTERNAL. Other errors during ORB initialization, such as failure to initialize a codeset manager, reactor, pluggable protocol factories, or default policies, will result in a CORBA::INITIALIZE exception. In all these
cases, the exception’s minor code will be set to
TAO_ORB_CORE_INIT_LOCATION_CODE.

0x0FU TAO_POLICY_NARROW_CODE
Not applicable.

0x10U TAO_GUARD_FAILURE
TAO’s POA and Real-Time POA maintain internal locks using the scoped
locking idiom, commonly known as a guard or lock monitor. If the POA or
the RT CORBA thread pool mechanism fails to acquire its internal lock via
the guard, a CORBA::INTERNAL exception will be raised with the minor code
set to TAO_GUARD_FAILURE.

0x11U TAO_POA_BEING_DESTROYED
An attempt to use the POA after it has been destroyed or as it is being
destroyed can result in a CORBA::BAD_INV_ORDER exception with the minor
code set to TAO_POA_BEING_DESTROYED.

0x12U TAO_POA_INACTIVE
An attempt to use the POA after it as been deactivated via its POA manager
will result in a CORBA::OBJ_ADAPTER exception with the minor code set to
TAO_POA_INACTIVE.

0x13U TAO_CONNECTOR_REGISTRY_INIT_LOCATION_CODE
If an invocation (such as the first invocation on an object reference) requires a
new connection, but initialization of the ORB core’s connector registry fails, a
CORBA::INITIALIZE exception will be raised with the minor code set to
TAO_CONNECTOR_REGISTRY_INIT_LOCATION_CODE.

0x14U TAO_AMH_REPLY_LOCATION_CODE
In an application using TAO’s Asynchronous Method Handling (AMH)
feature, if the response handler is deleted before a reply as been sent to the
client, a CORBA::NO_RESPONSE exception will be generated and sent to the
client. In other cases, if the application attempts to use a response handler
incorrectly, a CORBA::BAD_INV_ORDER exception will be raised. In both
cases, the minor code will be set to TAO_AMH_REPLY_LOCATION_CODE.

0x15U TAO_RTCORBA THREAD CREATION LOCATION_CODE
If an error occurs during thread creation in an RT CORBA thread pool, a
CORBA::INTERNAL exception will be raised with the minor code set to
TAO_RTCORBA THREAD CREATION LOCATION_CODE.
6.7.2 Error Number Codes

Many system exceptions are the result of a failure when accessing a system function. Most system functions set a global error number, known as `errno`, that identifies the reason for the failure. Table 6-3 gives the TAO preprocessor macro definition and a short description associated with each possible error number value included in the minor code.

Table 6-3 Minor Code Error Numbers

<table>
<thead>
<tr>
<th>Hex Value</th>
<th>Macro Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0U</td>
<td>TAO_UNSPECIFIED_MINOR_CODE</td>
<td>No error number associated with the exception.</td>
</tr>
<tr>
<td>0x1U</td>
<td>TAO_ETIMEDOUT_MINOR_CODE</td>
<td>Connection timed out.</td>
</tr>
<tr>
<td>0x2U</td>
<td>TAO_ENFILE_MINOR_CODE</td>
<td>System file table is full.</td>
</tr>
<tr>
<td>0x3U</td>
<td>TAO_EMFILE_MINOR_CODE</td>
<td>Process has too many open files.</td>
</tr>
<tr>
<td>0x4U</td>
<td>TAO_EPIPE_MINOR_CODE</td>
<td>No process to read the data from a pipe.</td>
</tr>
<tr>
<td>0x5U</td>
<td>TAO_ECONNREFUSED_MINOR_CODE</td>
<td>Target machine refused a connection.</td>
</tr>
<tr>
<td>0x6U</td>
<td>TAO_ENOENT_MINOR_CODE</td>
<td>A file or directory does not exist.</td>
</tr>
<tr>
<td>0x7U</td>
<td>TAO_EBADF_MINOR_CODE</td>
<td>A file descriptor refers to a file that is not open, or trying to read from a file opened only for writing.</td>
</tr>
<tr>
<td>0x8U</td>
<td>TAO_ENOSYS_MINOR_CODE</td>
<td>Operation not applicable.</td>
</tr>
<tr>
<td>0x9U</td>
<td>TAO_EPERM_MINOR_CODE</td>
<td>Not the super user.</td>
</tr>
<tr>
<td>0xAU</td>
<td>TAO_EAFNOSUPPORT_MINOR_CODE</td>
<td>Address not compatible with requested protocol.</td>
</tr>
<tr>
<td>0xBU</td>
<td>TAO_EAGAIN_MINOR_CODE</td>
<td>System process table is full.</td>
</tr>
<tr>
<td>0xCU</td>
<td>TAO_ENOMEM_MINOR_CODE</td>
<td>Not enough memory.</td>
</tr>
<tr>
<td>0xDU</td>
<td>TAO_EACCES_MINOR_CODE</td>
<td>File access denied.</td>
</tr>
<tr>
<td>0xEU</td>
<td>TAOEFAULT_MINOR_CODE</td>
<td>Attempting to access a bad address.</td>
</tr>
<tr>
<td>0xFU</td>
<td>TAO_EBUSY_MINOR_CODE</td>
<td>Device busy or lock is held.</td>
</tr>
<tr>
<td>0x10U</td>
<td>TAO_EEXIST_MINOR_CODE</td>
<td>File not expected to exist.</td>
</tr>
<tr>
<td>0x11U</td>
<td>TAO EINVAL_MINOR_CODE</td>
<td>An invalid argument was used.</td>
</tr>
<tr>
<td>0x12U</td>
<td>TAO_ECOMM_MINOR_CODE</td>
<td>Communication error on send.</td>
</tr>
<tr>
<td>0x13U</td>
<td>TAO_ECONNRESET_MINOR_CODE</td>
<td>Connection reset by peer.</td>
</tr>
<tr>
<td>0x14U</td>
<td>TAO_ENOTSUP_MINOR_CODE</td>
<td>Function not implemented.</td>
</tr>
</tbody>
</table>
6.8 Summary

- Exceptions provide a mechanism to communicate error information between CORBA clients and servers. Exceptions are mapped to either native C++ exceptions or to CORBA::Environment parameters.
- A set of CORBA::SystemException exceptions is defined for system-level errors.
- Domain- or application-specific exceptions are defined in IDL. The generated C++ exception classes inherit from CORBA::UserException.
- The ACE_TRY/ACE_CATCH family of macros enables the development of error-handling code that functions properly both in environments that support C++ exceptions and in environments that do not. A single code base can handle both situations, greatly simplifying cross-platform development.
- When a CORBA system exception is thrown, a minor code is provided to help identify the reason for the failure. Decoding this minor code can help in identifying the cause of the failure.
CHAPTER 7

CORBA Messaging

7.1 Introduction

The OMG introduced a CORBA Messaging specification (OMG Document formal/04-03-12, Chapter 22) to facilitate the development of portable CORBA code that efficiently supports the following:

- Making requests that do not require a client to block while waiting for a reply from a server. This is referred to as Asynchronous Method Invocation (AMI).
- Handling replies that are returned after the client process that submitted the associated request has terminated. This is referred to as Time Independent Invocation (TII).
- Allowing Quality of Service (QoS) to be specified for method invocations at the application level.

TAO 1.4a supports the callback model of AMI and a subset of the Messaging specification’s QoS policies. TAO 1.4a does not support Time-Independent Invocation of requests. The AMI callback model is described in 7.2. The QoS policies are described in 7.3. A Bi-Directional GIOP policy is described in 7.4.
Bi-Directional GIOP is defined in the CORBA Core specification (OMG Document formal/04-03-12) sections 15.8-15.9.

### 7.2 AMI Callback Model

#### The Problem
When a synchronous CORBA operation is invoked, the invoking client is blocked until it receives the reply to the operation request. A client cannot, however, always afford to spend its time exclusively waiting for the reply. Consider the following situations:

- **During time-consuming invocations to distributed objects, the user is not confident that the client is running properly.** While waiting for a reply, it may be desirable for the client application to provide periodic feedback to the user, verifying that the program is still waiting for a reply from the server.

- **A client wishes to make concurrent, rather than consecutive, requests to numerous servers.** Suppose that to obtain the total inventory count for a particular auto part, a number of warehouses must be queried. Whereas a single query may not be very time consuming, the cumulative waiting time for consecutive requests could be excessive. If all the queries are made concurrently, the total inventory count will be less time consuming.

- **The reply represents an event.** A client may want to be informed when some external event occurs, but does not want to block all other activities while waiting for such an event.

Prior to the communication models defined in the Messaging specification, the only models provided by CORBA for asynchronous communication were the *deferred synchronous* and *oneway* models.

Because the deferred synchronous model requires the use of the Dynamic Invocation Interface (DII), much more code is required for its implementation than for the synchronous model. In addition, the DII is tedious to use, not type-safe, and inefficient.

The oneway model requires the creation of a *reply-handler* object on the client side for handling replies to *oneway* operations that the client invokes on the server. An object reference to the reply handler is passed to the server in a *oneway* operation. To reply to a *oneway* request, the server must invoke an
operation on this reply handler. There are several disadvantages to this approach:

- The callback interface adds to the complexity of the IDL code.
- The interface for the server object must be altered to include the `oneway` definition and the callback object reference parameter.
- The server code needs to be written to also play the role of a client, so that it can invoke an operation on the callback object.
- Only the `oneway` operation is implemented. If the client application also needs a synchronous version of this operation, it must be defined separately.
- Traditional `oneway`s guarantee neither non-blocking semantics nor reliable delivery.

The AMI Callback Solution

The Asynchronous Method Invocation (AMI) callback\(^1\) model, defined by the Messaging specification and fully implemented in TAO 1.4a, addresses the above concerns by providing asynchronous operations that are not `oneway` operations and do not use the DII. These operations are referred to as `sendc_` operations throughout this chapter. A `sendc_` operation is provided to the client in addition to, rather than instead of, the corresponding synchronous operation, so the client may invoke either one at any time.

A `sendc_` operation has two purposes:

1. To cause the client ORB to send a request message to the server.
2. To provide the client ORB with an object reference to a reply handler.

AMI is a client-side language mapping issue, so enabling AMI does not alter the CORBA interface, and changes to server implementations are normally not required\(^2\). Thus, from the server’s perspective, a request message initiated by a `sendc_` operation invocation is identical to a request message initiated by the corresponding synchronous operation invocation.

The client ORB is the workhorse of the AMI callback model. It transforms the `sendc_` operation invocations into request messages and internally maps the request ID to the reply-handler object reference. When it receives a reply to a

---

1. An AMI polling model is also defined by the specification, but is not supported by TAO.
2. Servers may require changes to handle transactional asynchronous requests.
request message, it uses this internal mapping to invoke an operation on the
designated reply handler. A reply-handler skeleton class is implicitly
generated as part of the AMI callback model, so no explicit reply-handler
interface need be defined. The application developer simply writes a reply
handler that derives from this skeleton.

The AMI callback model is enabled by invoking the TAO IDL complier with
the -GC option. This produces the following additions to the resultant code for
each IDL interface:

1. A set of sendc_ member functions.
2. An ExceptionHolder class for delivering exceptions to the client.
3. A reply-handler skeleton class.

These components are described in 7.2.1 through 7.2.3. The process of writing
a reply handler is covered in 7.2.4. The chain of events that is set in motion by
invoking a sendc_ operation is illustrated in Figure 7-1 and described in
detail in 7.2.5.

The Messaging specification makes use of a concept called implied-IDL to
define the implementation of these components in a language-independent
manner. To distinguish it from actual IDL code, all implied-IDL code in this
chapter is displayed in italic type.

**Drawbacks to using AMI**

There are some drawbacks to using AMI that are worth noting:

- Additional stub code is generated for every operation and attribute in an
  interface. If only a small percentage of the operations and attributes are
  accessed using asynchronous invocations, and executable size is an issue,
  then it may make more sense to use alternative asynchronous techniques.

- Support for AMI callback is just beginning to appear in ORBs, so if client
  application portability among different ORB implementations is desired,
  it may be premature to deploy AMI-based application code. Since the use
  of AMI callback does not alter the CORBA interface, this is a client-side
  issue only. AMI callback clients are fully interoperable with non-AMI-
  callback servers.

- Client programmers have to write more code to support the AMI callback
  model than the normal synchronous invocation model. In particular, you
  must implement the reply-handler class, and you must supply an event
loop on the client side to handle asynchronous replies. Also, exception handling is considerably more complicated with the AMI callback model than with normal synchronous invocations.

7.2.1 Asynchronous sendc_ Operations

When invoked with the `-GC` option, the TAO IDL compiler generates `sendc_` member functions for the proxy\(^3\) (stub) class in addition to the synchronous member functions. These additional member functions can be thought of as being generated from implied-IDL operations\(^4\) that are added to the IDL interface. All `sendc_` operations have a return type of `void` and are defined as follows:

For each synchronous IDL operation `opName`, an implied-IDL operation named `sendc_opName` is defined according to the following rules:

- The first parameter is an `in` parameter named `ami_handler`, a reference to the designated reply handler.
- Each `in` and `inout` parameter in `opName` becomes an `in` parameter in `sendc_opName`.
- If `opName` has a `context` expression (specifying which elements of the client’s context may affect the performance of a request by the object), then `sendc_opName` will have an identical `context` expression.

The return value and `out` parameters of `opName` are ignored because they are handled by the reply handler.

For each IDL attribute `attrName`, an implied-IDL operation named `sendc_get_attrName` is defined. Its only parameter is an `in` parameter named `ami_handler`, a reference to its reply handler.

For each non-readonly IDL attribute `attrName`, an additional implied-IDL operation named `sendc_set_attrName` is defined according to the following rules:

- The first parameter is an `in` parameter named `ami_handler`, a reference to its reply handler.

3. See *Advanced CORBA Programming with C++*, 7.3 for more information about proxy classes.
4. These new operations are not considered to be real IDL because they do not correspond to entries in the Interface Repository.
• There is a second \texttt{in} parameter named \texttt{attrName} that has the same type as \texttt{attrName} and is used to set the attribute value.

If a \texttt{sendc} operation is invoked with a nil \texttt{ami\_handler} value, no response will be returned for that invocation.

Suppose we have the following IDL definition for \texttt{MyInterface}:

\begin{verbatim}
exception UserExcep {string usr_exc;};

interface MyInterface {
    boolean opName(in short a_short,
                    inout long a_long,
                    out float a_float)
        raises(UserExcep);
    attribute short attrib1;
    readonly attribute short attrib2;
};
\end{verbatim}

The implied-IDL \texttt{sendc} operations for \texttt{MyInterface} are:

\begin{verbatim}
void sendc\_opName(in AMI\_MyInterfaceHandler ami\_handler
                   in short a\_short,
                   in long a\_long);

void sendc\_get\_attrib1(in AMI\_MyInterfaceHandler ami\_handler);
void sendc\_set\_attrib1(in AMI\_MyInterfaceHandler ami\_handler,
                         in short attrib1);

void sendc\_get\_attrib2(in AMI\_MyInterfaceHandler ami\_handler);
\end{verbatim}

The \texttt{sendc} member functions of the C++ proxy (stub) class are:

\begin{verbatim}
virtual void sendc\_opName (AMI\_MyInterfaceHandler\_ptr ami\_handler,
                          CORBA::Short a\_short,
                          CORBA::Long a\_long);

virtual void sendc\_get\_attrib1 (AMI\_MyInterfaceHandler\_ptr ami\_handler);
virtual void sendc\_set\_attrib1 (AMI\_MyInterfaceHandler\_ptr ami\_handler,
                                 CORBA::Short attrib1);

virtual void sendc\_get\_attrib2 (AMI\_MyInterfaceHandler\_ptr ami\_handler);
\end{verbatim}
7.2.2 The ExceptionHolder

When a sendc_ operation is invoked, the client ORB attempts to send a request message to the server. If this attempt fails, the sendc_ operation raises a system exception with a completion status of COMPLETED_NO. Otherwise, the sendc_ operation returns normally and the client application continues.

If an exception occurs during the processing of a sendc_ request, the server returns this exception to the client ORB in the reply message, just as in the case of a synchronous operation. Unlike the synchronous case, however, the sendc_ operation cannot raise an exception because it returns before the reply is received.

When a reply to a sendc_ operation contains an exception, the client ORB receiving the reply must deliver this exception to the designated reply handler. Because CORBA exceptions cannot be passed as arguments in an IDL interface, the exception is inserted into an ExceptionHolder for delivery to the designated reply handler.

A C++ ExceptionHolder class is generated for each interface on which the TAO IDL compiler is invoked with the -GC option. This class can be thought of as having been compiled from an implied-IDL ExceptionHolder valuetype containing operations for raising the exceptions returned from sendc_ operations. The name of the corresponding implied-IDL ExceptionHolder valuetype for an interface named MyInterface is

AMI_MyInterfaceExceptionHolder

and the generated class name is

AMI_MyInterfaceExceptionHolder.

Each implied-IDL AMI_ *ExceptionHolder valuetype inherits from the Messaging::ExceptionHolder valuetype:

module Messaging
{
    valuetype ExceptionHolder {  
        public boolean is_system_exception;
        public boolean byte_order;
        public CORBA::OctetSeq marshaled_exception;
    };
};


AMI_*ExceptionHolder operations have no parameters and have a return type of void. They are defined as follows:

For each synchronous IDL operation opName, an implied-IDL operation named raise_opName is defined. If opName has a raises clause for user exceptions, then raise_opName has the same raises clause.

For each IDL attribute attrName, an implied-IDL operation named raise_get_attrName is defined.

For each non-readonly IDL attribute attrName, an additional implied-IDL operation named raise_set_attrName is defined.

Applying the above rules to MyInterface from 7.2.1 yields:

```c++
// ExceptionHolder implied-IDL
valuetype AMI_MyInterfaceExceptionHolder : Messaging::ExceptionHolder {
    void raise_opName() raises(UserExcep);
    void raise_get_attrib1();
    void raise_set_attrib1();
    void raise_get_attrib2();
};
```

The generated C++ AMI_*ExceptionHolder is:

```c++
class AMI_MyInterfaceExceptionHolder : public virtual Messaging::ExceptionHolder {
    public:
        virtual void raise_opName() = 0;
        virtual void raise_get_attrib1() = 0;
        virtual void raise_set_attrib1() = 0;
        virtual void raise_get_attrib2() = 0;
};
```

Note  TAO 1.4a implements the CORBA Messaging ExceptionHolder as described above. The TAO IDL Compiler generates a type-specific ExceptionHolder valuetype for each IDL interface as prescribed by the CORBA 2.6 specification (OMG Document formal/01-12-35). However, the CORBA 3.0.3 specification (OMG Document formal/04-03-12, 22.7) adds new operations to the interface of the CORBA Messaging ExceptionHolder. The new operations eliminate the need for a type-specific ExceptionHolder valuetype for each IDL interface. Instead, the generic Messaging::ExceptionHolder valuetype is extended with the ability to propagate a thrown exception through a new raise_exception()
7.2 AMI Callback Model

7.2.3 Reply Handler Operations

When the TAO IDL compiler is invoked with the `-GC` option, it generates a C++ reply-handler skeleton class for each interface on which it is invoked. This class can be thought of as having been compiled from an implied-IDL reply-handler interface. For an interface named `MyInterface`, the name of the implied-IDL interface is `AMI_MyInterfaceHandler` and the name of the generated skeleton class is `POA_AMI_MyInterfaceHandler`. The application developer writes a reply-handler class that inherits from the skeleton class and that is usually instantiated as a servant within the client.

However, the reply-handler servant does not have to be located within the client application. It can be located in another process. For example, if multiple instances of a client are instantiated, it may be desirable to handle all replies in only one of the instances. It is important to understand, however, that the client ORB that sends a request message will always receive the reply to this message. It is the client ORB that then invokes the reply-handler operation.

Reply-handler operations are invoked only by an ORB. They do not raise exceptions because they are never invoked by a client and thus have no client to respond to the exceptions. All reply-handler operations have a return type of `void` because their only purpose is to pass information to the reply handler.

An implied-IDL reply-handler interface contains two reply-handler operations for each `sendc_` operation, one to handle normal (non-exception) replies and another to handle exception replies. Thus there are two types of reply-handler operations:

- **Non-exception**: Delivers `sendc_` operation results.
- **Exception**: Delivers exceptions that occur during a `sendc_` operation.

**Non-Exception Replies**

Non-exception reply-handler operations are defined as follows:

For each IDL operation `opName`, an implied-IDL reply-handler operation named `opName` is defined according to the following rules:
If the operation has a return value, then the first parameter is an in parameter named \textit{ami\_return\_val} which is the return value of the IDL operation.

Each inout and out parameter in \textit{opName} (the IDL operation) becomes an in parameter in \textit{opName} (the reply-handler operation).

The in parameters of the IDL operation are ignored in the reply-handler operation because they are not part of the reply.

For an IDL attribute \textit{attrName}, the implied-IDL operation \textit{get\_attrName} is defined. It has a single in parameter named \textit{ami\_return\_val} of the same type as the attribute.

For a non-readonly IDL attribute \textit{attrName}, an additional implied-IDL operation named \textit{set\_attrName} with no parameters is defined.

There are two cases where the above rules will result in a reply-handler operation with no parameters:

- An IDL operation that has a return type of \textit{void} and no inout or out parameters.
- A non-readonly attribute (the set operation does not return a value).

In these cases, the reply-handler operation simply acknowledges a successful completion of the IDL operation.

Exception Replies

When an exception occurs during the processing of a \textit{sendc\_operation}, the exception is returned to the client ORB in the reply message, just as it is in the case of a synchronous operation. In the case of the \textit{sendc\_operation}, however, the client ORB inserts the exception into an \textit{ExceptionHolder} value and then invokes the designated reply-handler operation with this \textit{ExceptionHolder} as its parameter.

Reply-handler operations that deliver exceptions have a single in parameter named \textit{excep\_holder} and are defined as follows:

For each IDL operation \textit{opName}, an implied-IDL reply-handler operation named \textit{opName\_excep} is defined.

For an IDL attribute \textit{attrName}, an implied-IDL reply-handler operation named \textit{get\_attrName\_excep} is defined.
For a non-readonly IDL attribute attrName, an additional reply-handler implied-IDL operation named set_attrName_excep is defined.

Applying the above rules to MyInterface from 7.2.1 yields

```cpp
// Reply-handler implied-IDL
interface AMI_MyInterfaceHandler {
    void opName(in boolean ami_return_val,
                in long a_long,
                in float a_float);

    void opName_excep(in AMI_MyInterfaceExceptionHolder excep_holder);

    void get_attrib1(in short ami_return_val);
    void get_attrib1_excep(in AMI_MyInterfaceExceptionHolder excep_holder);

    void set_attrib1();
    void set_attrib1_excep(in AMI_MyInterfaceExceptionHolder excep_holder);

    void get_attrib2(in short ami_return_val);
    void get_attrib2_excep(in AMI_MyInterfaceExceptionHolder excep_holder);
};
```

The generated C++ reply-handler skeleton class for MyInterface is:

```cpp
class POA_AMI_MyInterfaceHandler: public virtual POA_Messaging::ReplyHandler {
public:

    // AMI callback exception support and TAO implementation code not shown.
    virtual void op(CORBA::Boolean ami_return_val,
                    CORBA::Long a_long,
                    CORBA::Float a_float) = 0;
    virtual void opName_excep(AMI_MyInterfaceExceptionHolder* excep_holder) = 0;

    virtual void get_attrib1(CORBA::Short ami_return_val) = 0;
    virtual void get_attrib1_excep(AMI_MyInterfaceExceptionHolder* excep_holder) = 0;

    virtual void set_attrib1() = 0;
    virtual void set_attrib1_excep(AMI_MyInterfaceExceptionHolder* excep_holder) = 0;

    virtual void get_attrib2(CORBA::Short ami_return_val) = 0;
    virtual void get_attrib2_excep(AMI_MyInterfaceExceptionHolder* excep_holder) = 0;
};
```
7.2.4 Creating a Reply-Handler Class

7.2.4.1 Generate Starter Code

When the options -GC (generate stub code for AMI callback support) and -GI (see 5.7) are simultaneously passed to the TAO IDL compiler, reply-handler class starter code is automatically generated for each interface in an IDL file. The two files generated are suffixed with I.h and I.cpp. For MyInterface, the generated files are MyInterfaceI.h and MyInterfaceI.cpp. Since these files also contain servant starter code that is only relevant to the server side, the reply-handler code should be extracted and placed in separate files. The following reply-handler code was extracted from MyInterfaceI.cpp and inserted into the file MyReplyHandler.cpp.

```c++
// ACE exception code is not shown
void AMI_MyInterfaceHandler_i::opName (CORBA::Boolean ami_return_val,
   CORBA::Long a_long,
   CORBA::Float a_float) {
   //Add your implementation here
}

void AMI_MyInterfaceHandler_i::opName_excep (AMI_MyInterfaceExceptionHolder* excep_holder) {
   //Add your implementation here
}

void AMI_MyInterfaceHandler_i::get_attrib1 (CORBA::Short ami_return_val) {
   //Add your implementation here
}

void AMI_MyInterfaceHandler_i::get_attrib1_excep (AMI_MyInterfaceExceptionHolder* excep_holder) {
   //Add your implementation here
}

void AMI_MyInterfaceHandler_i::set_attrib1 () {
   //Add your implementation here
}
```
void AMI_MyInterfaceHandler_i::set_attrib1_excep (AMI_MyInterfaceExceptionHolder* excep_holder)
{
   //Add your implementation here
}

void AMI_MyInterfaceHandler_i::get_attrib2 (CORBA::Short ami_return_val)
{
   //Add your implementation here
}

void AMI_MyInterfaceHandler_i::get_attrib2_excep (AMI_MyInterfaceExceptionHolder* excep_holder)
{
   //Add your implementation here
}

The AMI_MyInterfaceHandler_i reply-handler class inherits from the POA_AMI_MyInterfaceHandler class shown in 7.2.3. To complete the reply-handler class, the application developer needs to replace “Add your implementation here” with the desired functionality in each member function.

The type of functionality that is added to the starter code depends, first of all, on where the reply handler is to reside. Remember that the reply handler is not restricted to residing in the client process from which the sendc_ operation is invoked. It may reside in any process. The only restrictions are that the client must be able to obtain an object reference to the reply-handler object (to pass it in the sendc_ invocation) and the client process from which the operation was invoked must still be running when the reply is returned. (Remember that even if the reply handler is not part of the client application, the client ORB that invoked the request must receive the reply and invoke the reply-handler operation.)

In 7.2.4.2 and 7.2.4.3, we describe how to add the needed functionality to the reply-handler starter code.

### 7.2.4.2 Non-Exception Reply-Handler Functions

In general, if the reply handler resides within the client that invoked the sendc_ function, the reply handler copies the return value and other out parameters into appropriate variables and/or outputs a message. A common
way to store the return values and out parameters is to declare these variables as private members of the reply-handler class.

First, we show an example using non-AMI invocations. Using the MyInterface example again, the following client application makes use of the synchronous form of `opName` (assume that the IOR of the MyInterface object is stored in `my_interface.ior`):

```c++
#include "MyInterfaceC.h"

int main(int argc, char* argv[]) {
    try {
        CORBA::Boolean my_return_value;
        CORBA::Long my_long;
        CORBA::Float my_float;

        // Get an object reference to MyInterface object.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var obj = orb->string_to_object("file://my_interface.ior");
        MyInterface_var myInterface = MyInterface::_narrow(obj.in());

        my_return_value = myInterface->opName(10, my_long, my_float);

        // do other stuff...

        orb->destroy();
    } catch (CORBA::Exception&) {
        // Handle CORBA exceptions...
    }
}
```

In the above code segment:

- The return value of `opName` is stored in `my_return_value`.
- `opName`’s second variable, an inout variable, is stored in `my_long`.
- `opName`’s third variable, an out variable, is stored in `my_float`.

To achieve the same result using `sendc_opName`, first add the above three variables to MyReplyHandler.h as private data members of the AMI_MyInterfaceHandler_i class and add an accessor method for each. You may also need to add a mutual exclusion (mutex) lock to ensure thread-safe access to these private data members.
class AMI_MyInterfaceHandler_i : public virtual POA_AMI_MyInterfaceHandler
{
    public:
        AMI_MyInterfaceHandler_i (void);
        virtual ~AMI_MyInterfaceHandler_i (void);
        virtual void opName(CORBA::Boolean ami_return_val,
            CORBA::Long a_long,
            CORBA::Float a_float);
        virtual void opName_excep(AMI_MyInterfaceExceptionHolder* excep_holder);
        virtual void get_attrib1(CORBA::Short ami_return_val);
        virtual void get_attrib1_excep(AMI_MyInterfaceExceptionHolder* excep_holder);
        virtual void set_attrib1 ()
            virtual void set_attrib1_excep(AMI_MyInterfaceExceptionHolder* excep_holder);
        virtual void get_attrib2(CORBA::Short ami_return_val);
        virtual void get_attrib2_excep(AMI_MyInterfaceExceptionHolder* excep_holder);
    private:
        CORBA::Boolean get_my_return_value (void);
        CORBA::Long get_my_long (void);
        CORBA::Float get_my_float (void);
    private:
        CORBA::Boolean my_return_value_;
        CORBA::Long my_long_;
        CORBA::Float my_float_;

        ACE_Thread_Mutex lock_;
};

Now define the reply handler for opName() so that the reply variables are loaded into these private data members:

    ACE_Thread_Mutex AMI_MyInterfaceHandler_i::lock_;

    void AMI_MyInterfaceHandler_i::opName(CORBA::Boolean ami_return_val,
        CORBA::Long a_long,
        CORBA::Float a_float)
    {
        ACE_Guard<ACE_Thread_Mutex> guard(lock_);
        my_return_value_ = ami_return_val;
        my_long_ = a_long;
        my_float_ = a_float;
    }

These data members can be accessed within the client application as follows:
#include "MyReplyHandler.h"
#include "MyInterfaceC.h"

// Assume that -1 is not an allowable return value for my_long
// Initialize my_long to -1 so we can check later to see if it has changed
CORBA::Long AMI_MyInterfaceHandler_i::my_long = -1;

int main (int argc, char* argv[]) {
    try {
        // Get an object reference to MyInterface object.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var obj = orb->string_to_object("file://my_interface.ior");
        MyInterface_var myInterface = MyInterface::_narrow(obj.in());

        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
        PortableServer::POAManager_var mgr = poa->the_POAManager();
        mgr->activate();

        // Create a reply-handler servant.
        AMI_MyInterfaceHandler_i replyHandler_servant;
        PortableServer::ObjectId_var oid =
            poa->activate_object(&replyHandler_servant);
        CORBA::Object_var handler_obj = poa->id_to_reference(oid.in());
        AMI_MyInterfaceHandler_var replyHandler =
            AMI_MyInterfaceHandler::_narrow(handler_obj.in());

        // Invoke the operation asynchronously.
        myInterface_obj->sendc_opName(replyHandler.in(), 10);

        // do other stuff...

        while(1) {
            // Check to see if reply has been returned.
            if(orb->work_pending()) {
                orb->perform_work(); // Client ORB will invoke reply handler here
                // If the value of my_long has been changed, break out of while loop.
                if (replyHandler_servant.get_my_long() != -1) {
                    break;
                }
            }
        }

        orb->destroy();
    } catch (CORBA::Exception&) {
        // Handle CORBA exceptions...
    }
}
In the above client application, the ORB invokes the reply-handler function `opName()` on the client after the `orb->perform_work()` call is made from the client application. The result of this asynchronous client application differs from the synchronous one only in that the variables are private members of the reply-handler class. The variables will receive the same values in both cases.

### 7.2.4.3 Exception Reply-Handler Functions

We handle exceptions by adding `try` and `catch` blocks to each reply-handler exception function. Using `MyInterface` again in the example below, we add a `try` block and two `catch` blocks to the `opName` exception reply-handler:

```cpp
void AMI_MyInterfaceHandler_i::opName_excep (AMI_MyInterfaceExceptionHolder* excep_holder)
{
    try {
        excep_holder->raise_opName();
    }
    catch (CORBA::SystemException& e) {
        std::cout << "opName System Exception " << e << std::endl;
    }
    catch (CORBA::UserException& e) {
        std::cout << "opName user exception " << e.usr_exc << std::endl;
    }
}
```

The client ORB calls the `opName_excep()` reply handler when an exception is thrown during the processing of `opName()`. This exception is inserted into `AMI_MyInterfaceExceptionHolder` and passed to the `opName_excep()` reply-handler operation. The only way to gain access to this exception is to call the `AMI_MyInterfaceExceptionHolder` member function `raise_opName()`. This function throws the exception just as synchronous `opName()` does.

Calling the synchronous `opName()` from within a `try` block would have the same effect, as shown below:

```cpp
#include "MyInterfaceC.h"
#include <iostream>

int main (int argc, char* argv[])
{
    CORBA::Boolean return_value;

```
In the above code, the same exception is thrown and the same exception message is printed as in the asynchronous example.

### 7.2.4.4 Associating Replies with Requests

Before invoking a `sendc_` operation, a client must generate an object reference for the reply handler. In most cases, this object reference is generated once, then used repeatedly by the client application. However, there are situations where a client application needs to associate a unique identifier with each invocation of a `sendc_` operation so that it can distinguish between these requests at a later time. Also, there may be situations in which the client side needs to instantiate more than one instance of a reply handler.

Using one reply-handler instance to handle all replies coming from multiple server objects of the same type is technically correct, but not necessarily useful, since there is no way to distinguish callbacks resulting from AMI calls to different server objects. Here are some common strategies for addressing this problem:

- **Servant-per-AMI-call strategy**: This strategy involves the instantiation and activation of a separate reply-handler instance for each AMI call. The drawback, of course, is that if there are many simultaneous asynchronous calls, the memory footprint of the client will increase. This strategy is simpler to implement programmatically than the activation-per-AMI strategy and results in less data being marshaled/demarshaled and sent over the wire than the server-differentiated-reply strategy.

- **Activation-per-AMI-call strategy**: One way to distinguish separate AMI calls without using a separate reply-handler instance for each invocation is
to explicitly activate the same servant multiple times in the client’s POA. Using the `PortableServer::IdAssignmentPolicy` of `MULTIPLE_ID` with a non-root POA, you can activate a servant multiple times, each time with a different user-chosen object id. The reply-handler callback methods examine this object id to determine which request caused the reply.

Before making AMI calls, the application creates a POA with the `MULTIPLE_ID` and `USER_ID` policies. For each AMI call, the application creates a special object id and maps the object id to the reply-handler servant using the `PortableServer::activate_object_with_id()` operation. After the client makes the AMI call, the reply arrives at the reply handler. The reply-handler uses the `PortableServer::Current` interface to obtain the object id and associate the reply with the correct request. See *Advanced CORBA Programming with C++* 11.4.2 and 11.4.3 for more information on the `USER_ID` and `MULTIPLE_ID` policies and *Advanced CORBA Programming with C++* 11.7.4 for more information on the `PortableServer::Current`.

Although this approach is more complex to implement, it is more scalable than the Servant-per-AMI-call strategy because it uses a single servant for all asynchronous calls. However, both strategies require an entry-per-AMI-call in the client POA’s active object map. One way to reduce this overhead is to use a Servant Locator that activates the client’s reply handler on demand, thereby minimizing memory utilization. The activation-per-AMI-call has the advantage over the Server-differentiated-reply strategy of reduced marshaling/demarshaling and less data traveling over the wire.

- **Server-differentiated-reply strategy**: This strategy provides an alternative for differentiating multiple AMI calls, but requires a small modification to the IDL interface. An `out` parameter is added to the function signature for use by the server side to add information that will assist the client-side reply handler in distinguishing which reply goes with which request. Thus, just one servant is needed for distinguishing between all AMI callbacks, and it only needs to be activated once in the client’s POA.

However, compared to allocating a different servant for each AMI call, the use of an `out` parameter is obtrusive and incurs more network overhead to pass the added parameter back to the client. The network overhead can be limited by using the Asynchronous Completion Token
(ACT) pattern of adding a small, fixed-size inout parameter to the function call. The ACT is first initialized by the client to indicate a particular AMI call and then passed to the server. The server subsequently returns the ACT unchanged as a parameter to the reply handler callback. The reply handler maps the ACT to the associated actions and states necessary to complete the reply processing. If the size of the ACT is smaller than the out parameter described in the earlier part of this strategy, the network bandwidth consumption is reduced somewhat.

The above strategies are further described, with examples, in

7.2.5 The Processing of an AMI sendc_operation
When an AMI sendc_operation is invoked, the following sequence of steps is initiated (See Figure 7-1):

1. The client invokes an AMI sendc_operation on its server object reference.
2. The object reference passes the request to the client ORB.
3. The client ORB:
   - Assigns a unique ID number to the request.
   - Creates a mapping between the ID and the reply-handler object reference.
   - Packages the request message and hands it off to the OS.
4. The client OS sends the request message to the server.
5. The server OS stores the request in the server ORB’s message buffer.
6. The server ORB:
   - Gets the request from the message buffer.
   - Unpacks the message.
   - Invokes the synchronous operation on the servant.
7. The servant processes the operation and returns the reply to the ORB.
8. The server ORB packages the reply message and hands it off to the OS.
9. The server OS sends the reply message to the client.
10. The client OS stores the reply in the client ORB’s message buffer.
Because the `sendc_` operation is asynchronous, the client is able to process other tasks while steps 2 through 10 are taking place. At some point after invoking the `sendc_` operation (unless the client is multithreaded and the ORB is running in its own thread, or the reply handler is not located in the client application), the client must invoke either `perform_work()` or `run()` on the ORB to retrieve the reply. The following sequence of steps is then initiated:

11. The client invokes either `run()` or `perform_work()` on the client ORB.
12. The client ORB:
   - Gets the reply from the message buffer.
   - Unpacks the reply message and extracts the ID number.
   - Uses the ID number to locate the designated reply-handler object reference.
   - Invokes the appropriate reply-handler operation on the reply-handler object reference.
13. The client processes the reply-handler operation.
7.2.6 AMI Callback Example

Now that you know how to use a `sendc_` operation and write a reply handler, we show a complete example that uses the AMI callback feature of TAO.

The example shown here is based on the Messenger example, first introduced in Chapter 3. Full source code for this example is in the TAO 1.4a source code distribution in the directory `$TAO_ROOT/DevGuideExamples/Messaging/AMIcallback`.

7.2.6.1 IDL Definitions

The IDL file used for the Messenger is shown below:

```idl
// Messenger.idl

exception MessengerUnableToSendMessage
```
In the above code, the string message parameter is an `inout` parameter since only part of the message may get displayed (on a pager, for example). The returned value in this case is the partial message sent to the user.

The implied-IDL for the reply handler for the Messenger interface is:

```java
interface AMI_MessengerHandler
{
    void send_message(in boolean ami_return_val,
                      in string message,
                      in long time_sent);

    void send_message_excep(in AMI_MessengerExceptionHolder excep_holder);
};
```

The Messenger interface with the implied-IDL `sendc_` operations included is as follows:

```java
interface Messenger
{
    boolean send_message(in string user_name,
                          in string subject,
                          inout string message,
                          out long time_sent)
        raises (MessengerUnableToSendMessage);

    void sendc_send_message(in AMI_MessengerHandler ami_handler,
                            in string user_name,
                            in string subject,
                            in string message);
};
```
7.2.6.2 Generating Starter Implementation Code

To minimize the code generated by the IDL compiler, AMI callback stub code is not generated by default. Therefore, we need to inform the IDL compiler to generate this code by passing the `-GC` option. To minimize the amount of code we need to write, we tell the compiler to generate starter implementation code by using the `-GIh` and `-GIs` options. The resulting command line appears as follows:

```
tao_idl -GC -GIh _i.h -GIs _i.cpp Messenger.idl
```

After this command is run, the starter code for the Messenger servant and AMI reply-handler implementations can be found in the files `Messenger_i.h` and `Messenger_i.cpp`. Invoke the IDL compiler manually (instead of through a Makefile) to avoid overwriting implementation code that you have added to the generated starter code.

Since the AMI reply-handler code is for the client side only, we remove the reply-handler starter code from `Messenger_i.h` and `Messenger_i.cpp`, then place it into files named `MessengerHandler.h` and `MessengerHandler.cpp`, respectively.

7.2.6.3 The Messenger Servant Code

To help illustrate the usage of AMI, we add a private data member called `seconds_to_wait_` to the `Messenger_i` class defined in `Messenger_i.h`. By having the server artificially wait `seconds_to_wait_` seconds before it sends the reply, we can mimic the effects of an actual server that may take a while to send a reply.

In addition, `Messenger_i` has a `CORBA::Boolean` data member called `throw_exception_` that the `send_message()` implementation uses to force an exception to be thrown, thus allowing the client-side exception handling code to be exercised.

The constructor for `Messenger_i` accepts arguments to initialize `seconds_to_wait_` and `throw_exception_`. These arguments are set based on command-line arguments passed to the server executable (see `$TAO_ROOT/DevGuideExamples/Messaging/AMIcallback/MessengerServer.cpp` in the TAO 1.4a source code distribution to see how this is done).
The send_message() member function is shown below (the code in bold has been added to the IDL-compiler-generated starter code):

```cpp
CORBA::Boolean Messenger_i::send_message ( 
    const char * user_name, 
    const char * subject, 
    char *& message, 
    CORBA::Long_out time_sent 
)
throw (CORBA::SystemException, MessengerUnableToSendMessage) 
{
    if (throw_exception_)
    {
        std::cout << "Throwing MessengerUnableToSendMessage exception." << std::endl;
        throw MessengerUnableToSendMessage();
    }

    std::cout << "Write a letter to " << user_name << " as follows:" << std::endl;
    std::cout << "Subject: " << subject << std::endl;
    std::cout << "Dear " << user_name << "," << std::endl;
    std::cout << message << std::endl;

    if (seconds_to_wait_ > 0)
    {
        std::cout << "Waiting for " << seconds_to_wait_ << " seconds..." << std::flush;
        ACE_OS::sleep(seconds_to_wait_);
        std::cout << " Done waiting" << std::endl;
    }

    // Record the time the message was sent
    time_sent = ACE_OS::gettimeofday().sec();

    // We will assume the message has been sent, so return true
    return 1;
}
```

### 7.2.6.4 The Reply-Handler Class Definition

For this example, the MessengerHandler reply-handler echoes the server’s response to standard output, including the time the message was sent. It shuts down the ORB after one message.

The reply-handler functions related to the send_message() operation are shown below:

```cpp
void MessengerHandler::send_message ( 
    CORBA::Boolean ami_return_val, 
    const char * message, 
```
CORBA Messaging

CORBA::Long time
)
throw (CORBA::SystemException)
{
    if (ami_return_val)
    {
        time_ = time;
        time_t t = time_;
        const char * time_str = ACE_OS::ctime(&t);
        if (time_str != 0) {
            std::cout << std::endl << "Message sent at " << time_str << std::endl;
        }
        std::cout << "Content of message: " << message << std::endl;
    } else
    {
        std::cerr << "Error: Message was not sent." << std::endl;
    }
    // Our simple test just shuts down after sending one message.
    orb_->shutdown(0);
}

void MessengerHandler::send_message_excep (AMI_MessengerExceptionHolder* excep_holder
) throw (CORBA::SystemException)
{
    // We'll print an error message and shut down the orb
    try
    {
        excep_holder->raise_send_message();
    }
    catch (CORBA::Exception& ex)
    {
        std::cerr << "A CORBA Exception was thrown: " << ex << std::endl;
    }
    orb_->shutdown(0);
}

7.2.6.5 The Client Code

Since the reply handler will be called back by the ORB when the reply arrives from the server, it needs to be registered with the POA as a servant just like servants are registered in server code. The client code is:

#include "MessengerC.h"
#include "MessengerHandler.h"
# 7.2 AMI Callback Model

```cpp
int
main(int argc, char * argv[])
{
  try {

    // assume any command line parameter means we want an automated test.
    bool automated = argc > 1;

    // Initialize orb
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    CORBA::Object_var obj = orb->string_to_object("file://MessengerServer.ior");
    if (CORBA::is_nil(obj.in())) {
      std::cerr << "Nil Messenger reference" << std::endl;
      return 1;
    }

    // Narrow
    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (CORBA::is_nil(messenger.in())) {
      std::cerr << "Argument is not a Messenger reference" << std::endl;
      return 1;
    }

    // Get reference to Root POA.
    obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

    // Activate POA manager
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Register an AMI handler for the Messenger interface
    MessengerHandler servant(orb.in());
    PortableServer::ObjectId_var oid = poa->activate_object(&servant);
    obj = poa->id_to_reference(oid.in());
    AMI_MessengerHandler_var handler = AMI_MessengerHandler::_narrow(obj.in());

    For our example, we will get the necessary information needed to send a
    message to a particular user from standard input.

    CORBA::String_var user = CORBA::string_alloc(81);
    CORBA::String_var subject = CORBA::string_alloc(81);
    CORBA::String_var message = CORBA::string_alloc(81);

    if (! automated) {
      std::cout << "Enter user name -->";
      std::cin.getline(user, 81);
    }

    // ...
std::cout << "Enter subject -->";
std::cin.getline(subject, 81);

std::cout << "Enter message -->";
std::cin.getline(message, 81);
} else {
    user = CORBA::string_dup("TestUser");
    subject = CORBA::string_dup("TestSubject");
    message = CORBA::string_dup("Have a nice day.");
}

// Record the time the request was made.
ACE_Time_Value time_sent = ACE_OS::gettimeofday();

messenger->sendc_send_message(handler.in(),
    user.in(),
    subject.in(),
    message.in());

Now we will run an event loop that runs the ORB in a non-blocking fashion. Doing so allows us to provide feedback to the user when control is handed back to the main thread.

Note  
See Advanced CORBA Programming with C++, 11.11.2, for details on performing non-blocking event handling with CORBA.

// Do some work to prove that we can send the message asynchronously, then come back later and retrieve the results.

for (int i = 0; i < 10; ++i) {
    ACE_OS::printf(".");
    ACE_OS::sleep(ACE_Time_Value(0, 10 * 1000));
}

// Our simple servant will exit as soon as it receives the results.
orb->run();

if (servant.message_was_sent())
{
    // Note: We cannot use the time sent by the server to compare with
    // the time value here in the client because the server machine's
    // clock may not be synchronized with the client's clock.

    ACE_Time_Value delay = ACE_OS::gettimeofday() - time_sent;
    std::cout << std::endl << "Reply Delay = "
              << delay.msec() << "ms" << std::endl;
7.2 AMI Callback Model

7.2.6.6 Building Applications that use AMI

TAO’s support of AMI is implemented in the TAO_Messaging library. Thus, applications that use AMI must link with this library. MPC projects for applications that use AMI can simply inherit from the messaging and ami base projects. For example, below is the MPC file for the AMI callback example in $TAO_ROOT/DevGuideExamples/Messaging/AMIcallback:

```mpc
project(*Server): messaging, taoexe, portableserver, ami {
    requires += exceptions
    Source_Files {
        Messenger_i.cpp
        MessengerServer.cpp
    }
}

project(*Client): messaging, taoexe, portableserver, ami {
    requires += exceptions
    Source_Files {
        MessengerHandler.cpp
        MessengerClient.cpp
    }
}
```

For more information on MPC, see Chapter 4.

7.2.7 Controlling the Delivery of AMI-based Requests

TAO’s implementation of AMI permits the Messaging SyncScope policy to be applied to the delivery of requests that use AMI. This feature allows fine-grained control over the ORB’s return of control back to the client application code. See 7.3.4 for more information on the Messaging SyncScope policy. In addition, the TAO-specific buffering constraint policy can be applied to specify the conditions under which a queue of requests should be buffered and
transmitted. See 7.3.5 for more information on TAO’s buffering constraint policy.

7.3 Quality of Service Policies

Quality of Service (QoS) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that application developers only need to specify what they want rather than how it should be achieved.

Generally speaking, quality of service comprises several QoS policies. Each policy is an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to greater flexibility in application design.

The CORBA Messaging specification defines mechanisms by which clients and servers can set required and supported qualities of service with respect to requests. It describes a standard QoS framework within which CORBA services can define their service-specific qualities. In this framework, all QoS settings are local interfaces derived from CORBA::Policy. Many of these QoS policies are defined in the Messaging module in IDL. TAO defines some additional QoS policies, also derived from CORBA::Policy, that fit within the CORBA Messaging QoS framework.

The following sections describe the messaging QoS policies supported by TAO and how to use them.

7.3.1 Policy Management

7.3.1.1 Creating Policies

As stated above, all CORBA Messaging QoS policies inherit from CORBA::Policy (the same base interface used to specify POA policies). CORBA::ORB has a generic factory operation, create_policy(), that can be used to create new policy objects. This operation is defined in the following IDL:

module CORBA {
    typedef unsigned long PolicyType;
    interface Policy {};
typedef short PolicyErrorCode;
const PolicyErrorCode BAD_POLICY = 0;
const PolicyErrorCode UNSUPPORTED_POLICY = 1;
const PolicyErrorCode BAD_POLICY_TYPE = 2;
const PolicyErrorCode BAD_POLICY_VALUE = 3;
const PolicyErrorCode UNSUPPORTED_POLICY_VALUE = 4;
exception PolicyError {PolicyErrorCode reason;};

interface ORB { Policy create_policy(in PolicyType type, in any val) raises(PolicyError); };

Each messaging QoS policy is assigned a unique PolicyType. For example, the policy type for the relative round-trip timeout policy, described in 7.3.2, is defined as:

module Messaging {
  const CORBA::PolicyType RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
};

The val parameter passed to create_policy() is a CORBA::Any that contains the desired value for the policy. If the CORBA::Any does not contain an acceptable value or if the CORBA::PolicyType is not supported by the ORB, a CORBA::PolicyError exception is raised.

Before policies can be applied, they must be added to a CORBA::PolicyList. For example:

// Create a policy and add it to a CORBA::PolicyList.
CORBA::Any policy_value_as_any;
// initialize "policy_value_as_any" with a value
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] = orb->create_policy (SOME_POLICY_TYPE, policy_value_as_any);

On the client side, policies are applied to various objects, such as the ORB, the current thread of execution, or a specific object reference. On the server side, policies are applied to the POA.
### 7.3.1.2 Client Side Policy Management

Messaging QoS policies can be applied on the client side at three different scoping levels. This permits you to work with a level of granularity that is appropriate for your application. These levels are as follows:

1. **The ORB level.** Policies applied at the ORB level will apply to all requests delivered by the ORB. The CORBA::PolicyManager is used to set policies at this level. For example:

   ```
   CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
   CORBA::PolicyManager_var policy_manager =
       CORBA::PolicyManager::_narrow(obj.in());
   policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);
   ```

2. **The thread level.** Using thread-level policies allows quality-of-service values to be applied to operations invoked from a certain thread. For example:

   ```
   CORBA::Object_var obj = orb->resolve_initial_references("PolicyCurrent");
   CORBA::PolicyCurrent_var policy_current =
       CORBA::PolicyCurrent::_narrow (obj.in());
   policy_current->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);
   ```

3. **The object reference level.** For the most fine-grained level of control, quality-of-service policies can be set on a per-object-reference basis. Assuming we have some object reference `obj`, we can apply policies to the object reference as follows:

   ```
   CORBA::Object_var new_obj =
       obj->_set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
   ```

   Note that CORBA::Object::_set_policy_overrides() returns an object reference that you must narrow to a specific interface type before invoking operations on it. It does *not* modify the object reference upon which it is called. In the above example, the `new_obj` object reference contains the new `policy_list` policies.

Policy overrides applied at the object-reference level take precedence over those applied at the thread or ORB level. Likewise, policy overrides applied at the thread level take precedence over those applied at the ORB level.

Whether new policy settings are *added to* or *replace* existing policy settings is controlled by the second parameter in `set_policy_overrides()`. If the second parameter is CORBA::SET_OVERRIDE, the policies in the policy list
completely replace the existing policies set at the relevant level of granularity. If the second parameter is `CORBA::ADD_OVERRIDE`, the new policies are added to the existing policies, unless a given policy in effect has the same `PolicyType` as one of the policies in the list, in which case the new policy replaces the existing policy.

### 7.3.1.3 Server Side Policy Management

On the server side, policies are associated with a POA. Policies that are applicable to server-side behavior can be passed via a `CORBA::PolicyList` to the `POA::create_POA()` operation. Request processing through a POA is subject to the policies applied to that POA at its creation. Some policies applied to a POA are exported to clients via object references created through that POA.

For more information on POA creation and POA policies, see *Advanced CORBA Programming with C++*, Chapter 11.

### 7.3.1.4 Destroying Policies

Once the policies have been applied at either the client side or the server side, the policy objects themselves should be destroyed using the `CORBA::Policy::destroy()` operation. For example:

```cpp
// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
    policy_list[i]->destroy();
}
policy_list.length(0);
```

### 7.3.2 Request and Reply Timeouts

The CORBA Messaging specification defines a relative round-trip timeout policy. Relative round-trip timeouts are used to limit the total amount of time spent completing the following steps:

1. The client attempts to make a connection with a server.
2. The client passes a request to the server.
3. The client waits for a reply from the server.

At each step, the time spent since the start of the request is checked against a user-specified timeout value. If the time exceeds this timeout value a
CORBA::TIMEOUT exception is raised. Specifying a relative round-trip timeout value is useful in real-time and fault-tolerant systems, since the client can take appropriate action if the server becomes unresponsive or cannot complete a request within a specified time interval.

Only clients are impacted by the use of the relative round-trip timeout policy; no timing requirements are passed to the server. If a CORBA::TIMEOUT exception is raised and a server reply arrives sometime afterward, the reply is simply ignored.

The PolicyType for the relative round-trip timeout policy is Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE. Its value is a CORBA::Any containing a TimeBase::TimeT as defined in the CORBA Time Service specification. The timeout value is a 64-bit value interpreted as hundreds of nanoseconds. If either the CORBA::Any reference does not contain a TimeBase::TimeT or the CORBA::PolicyType is not supported, then a CORBA::PolicyError exception is raised.

The following code shows how to create a relative round-trip timeout policy with a timeout value of one millisecond, and apply it at the ORB level. A complete example showing how to use this policy is included in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/Messaging/RelativeRoundTripTimeout.

```c++
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Set the policy value to 1 millisecond (10 * 1000 msec/ usec).
TimeBase::TimeT relative_rt_timeout = 10000;  // 1 millisecond
CORBA::Any relative_rt_timeout_as_any;
relative_rt_timeout_as_any <<= relative_rt_timeout;

// Create the policy and add it to a CORBA::PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] =
orb->create_policy (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE,
relative_rt_timeout_as_any);

// Apply the policy at the ORB level.
CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);
```
// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
    policy_list[i]->destroy ();
}
policy_list.length(0);

A client that invokes a request on a server while a relative timeout policy is in effect may receive an exception of type CORBA::TIMEOUT. This exception is generated by the underlying invocation implementation upon expiration of the specified time limit. Prior to the exception being thrown, the request is cancelled if a response has not yet been received from the server.

---

**Note** In addition to relative round-trip timeouts, five additional timeout policies are defined in the CORBA Messaging specification. These additional timeout policies apply to request start time, request end time, reply start time, reply end time, and relative request delivery time. None of these is available in TAO 1.4a. However, TAO 1.4a does provide a TAO-specific connection timeout policy, described in 7.3.3.

---

### 7.3.3 Connection Timeouts

In addition to the relative round-trip timeout policy described in 7.3.2, TAO provides a policy to control connection timeouts. The connection timeout policy is used to limit the total amount of time a client spends establishing a connection with a server. If the connection time exceeds the value specified in the policy, a CORBA::TIMEOUT exception is raised. Specifying a connection timeout value is useful in real-time and fault-tolerant systems, since the client can take appropriate action if the server becomes unresponsive or if a network interruption occurs.

The TAO connection-timeout-policy local interface is defined in $TAO_ROOT/tao/Messaging/TAO_Ext.pidl as follows:

```plaintext
#pragma prefix "tao"

module TAO
{
    const CORBA::PolicyType CONNECTION_TIMEOUT_POLICY_TYPE = 0x54410008;

    local interface ConnectionTimeoutPolicy : CORBA::Policy {
        readonly attribute TimeBase::TimeT relative_expiry;
    };
}
```
The following example shows how to set the ConnectionTimeoutPolicy to 200 milliseconds on an object reference. The example uses the CORBA Messaging validate_connection() operation to explicitly open the connection and verify that the connection can be made within the specified timeout.

```cpp
try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Set the policy value (1.0e-3 * 1.0e7 is 1 millisecond).
    TimeBase::TimeT connection_timeout = 1.0e-3 * 1.0e7 * 200;
    CORBA::Any connection_timeout_as_any;
    connection_timeout_as_any <<= connection_timeout;

    // Create the policy and add it to a CORBA::PolicyList.
    CORBA::PolicyList policy_list;
    policy_list.length(1);
    policy_list[0] =
        orb->create_policy (TAO::CONNECTION_TIMEOUT_POLICY_TYPE,
                            connection_timeout_as_any);

    // Obtain an object reference.
    CORBA::Object_var obj = orb->string_to_object("file://MessengerServer.ior");

    // Apply the policy to the object reference; returns a new object reference.
    CORBA::Object_var new_obj =
        obj->_set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);

    // Destroy the Policy objects.
    for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
        policy_list[i]->destroy();
    }
    policy_list.length(0);

    // Explicitly bind a connection to the server (may timeout).
    CORBA::PolicyList_var inconsistent_policies;
    CORBA::Boolean status =
        new_obj->_validate_connection (inconsistent_policies.out());

    // _narrow() and use the new_obj object reference as usual...
}

catch (CORBA::TIMEOUT&) {
    // The connection attempt timed out.
}
```
catch (CORBA::Exception&) {
    // Some other CORBA exception was raised.
}

You can see another example of using the connection timeout policy in $TAO_ROOT/tests/Connection_Timeout. See the README file in that directory for more information.

---

**Note**  
*The connection timeout policy is specific to TAO. It is not part of the CORBA Messaging specification.*

---

### 7.3.4 Reliable Oneway calls using the SyncScope Policy

The CORBA Messaging specification defines a policy called *SyncScope* that is implemented in TAO 1.4a. This policy permits clients to specify at what stage during a oneway message invocation control is returned back to the client application code. There are four possible values that can be used for the SyncScope policy that are given below.

- **SYNC_NONE** Using this policy value causes the client ORB to return control to the client application before the request is passed to the transport protocol. For this case the client is guaranteed not to block during a request invocation. This policy value provides the lowest guarantee of delivery.

- **SYNC_WITH_TRANSPORT** Setting the *SyncScope* policy to this value causes control to return to the client application code after the transport has accepted the request. Use of this policy value does not guarantee that the request has been delivered to the server. For example, if IIOP is being used, then limited TCP buffer space may cause unbounded delays in transmission. **SYNC_WITH_TRANSPORT** is the default SyncScope policy value in TAO.

- **SYNC_WITH_SERVER** When the *SyncScope* policy is set to this value, the server sends its reply before invoking the target servant. This setting is useful if the reliability of the network is of concern and the time spent executing the servant code dominates the time involved in waiting for a reply. The stage at which the server sends back an acknowledgement is right after the use of any servant manager, but before the target servant is invoked.
• **SYNC_WITH_TARGET** This policy value has the same effect as turning a oneway call into a synchronous call by removing the oneway qualifier in the operation signature. Control is returned to a client application only after the reply has been received from the target servant. Use this policy value if you need complete confidence that a reply has been received from the server and only if it is appropriate for the client application to block while the target servant is preparing a reply.

Both the **SYNC_NONE** and **SYNC_WITH_TRANSPORT** policy values are valid interpretations of the original oneway semantics defined by the CORBA specification.

The following example shows how to set the SyncScope policy such that oneway invocations do not return control to the client until the client has received an acknowledgement from the server that the message has been delivered to the servant. In this example, we apply the policy at the ORB level:

```cpp
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Set the policy value.
Messaging::SyncScope sync_with_target = Messaging::SYNC_WITH_TARGET;
CORBA::Any sync_with_target_any;
sync_with_target_any <<= sync_with_target;

// Create the policy and add it to a CORBA::PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] =
    orb->create_policy (Messaging::SYNC_SCOPE_POLICY_TYPE, sync_with_target_any);

// Apply the policy at the ORB level.
CORBA::Object_var obj = orb->resolve_initial_references ("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
    CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);

// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
    policy_list[i]->destroy ();
}
policy_list.length(0);

// ... rest of application ...
```
7.3 Quality of Service Policies

Figure 7-2 shows the effects of the various settings of the SyncScope policy on oneway invocations.

7.3.5 Buffered Oneway and Asynchronous Requests

TAO provides a BufferingConstraint policy to control the dispatching of oneway and asynchronous requests from a client. This policy is not part of the CORBA Messaging specification, but uses the same QoS policy framework as the other policies described in this chapter. Using the BufferingConstraint policy, it is possible to specify that oneway and asynchronous requests should be buffered in the client’s ORB and dispatched only when one or more of the following conditions, controllable via the policy value, has been met:

- A specified timeout value has expired.
- A specified maximum message byte count has been reached.
- A specified maximum message count has been reached.
- An explicit buffer flush has been issued.
- The ORB has been shut down.

By default, oneway and asynchronous requests are not buffered. To set the BufferingConstraint policy, you create and initialize a structure of type TAO::BufferingConstraint to describe how request buffering is to be performed. The BufferingConstraintPolicy interface, BufferingConstraint structure, and related constants are defined in $TAO_ROOT/tao/TAO.pidl as follows:

```plaintext
#pragma prefix "tao"

module TAO {

typedef unsigned short BufferingConstraintMode;
const BufferingConstraintMode BUFFER_FLUSH = 0x00;

// Note that timeout, message_count, and message_bytes can be or'd.
const BufferingConstraintMode BUFFER_TIMEOUT = 0x01;
const BufferingConstraintMode BUFFER_MESSAGE_COUNT = 0x02;
const BufferingConstraintMode BUFFER_MESSAGE_BYTES = 0x04;

struct BufferingConstraint {
    BufferingConstraintMode mode;
    TimeBase::TimeT timeout;
    unsigned long message_count;
    unsigned long message_bytes;
};

const CORBA::PolicyType BUFFERING_CONSTRAINT_POLICY_TYPE = 0x54410001;
local interface BufferingConstraintPolicy : CORBA::Policy {
    readonly attribute BufferingConstraint buffering_constraint;
};
}
```

To initialize the mode data member of the BufferingConstraint structure, compute the bitwise OR of one or more TAO::BufferingConstraintMode constants (e.g., TAO::BUFFER_TIMEOUT|TAO::BUFFER_MESSAGE_COUNT). Depending upon the value of mode, one or more of the timeout, message_count, or message_bytes data members should also be set.
Note that buffering of oneway requests requires the SyncScope policy, described in 7.3.4, to be specified as Messaging::SYNC_NONE.

The following example shows how to set the BufferingConstraint policy such that oneway and asynchronous invocations are buffered in the client ORB until a particular message count or total buffer size has been reached. In this example, we apply the policy at the ORB level:

```cpp
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Set the SyncScope policy for oneways to SYNC_NONE.
Messaging::SyncScope sync_none = Messaging::SYNC_NONE;
CORBA::Any sync_none_any;
sync_none_any <<= sync_none;

// Set the BufferingConstraint policy to buffer up to 5 requests
// or until a total of 4K bytes have been buffered.
TAO::BufferingConstraint buffering_constraint;
buffering_constraint.mode =
    TAO::BUFFER_MESSAGE_COUNT | TAO::BUFFER_MESSAGE_BYTES;
buffering_constraint.message_count = 5;
buffering_constraint.message_bytes = 4096;
buffering_constraint.timeout = 0;
CORBA::Any buffering_constraint_any;
buffering_constraint_any <<= buffering_constraint;

// Create the policies and add them to a CORBA::PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(2);
policy_list[0] =
    orb->create_policy (Messaging::SYNC_SCOPE_POLICY_TYPE, sync_none_any);
policy_list[1] = orb->create_policy (TAO::BUFFERING_CONSTRAINT_POLICY_TYPE,
                                           buffering_constraint_any);

// Apply the policies at the ORB level.
CORBA::Object_var obj = orb->resolve_initial_references ("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
    CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);
```

You can see another example using the BufferingConstraint policy with oneway requests in $TAO_ROOT/tests/Oneway_Buffering. You can see an example of using the BufferingConstraint policy with asynchronous
(AMI) requests in `$TAO_ROOT/tests/AMI_Buffering`. See the README file in each of those directories for more information.

**Note**  
The buffering constraint policy is specific to TAO. It is not part of the CORBA Messaging specification.

### 7.3.5.1 Building Applications that use Messaging QoS

TAO’s support of Messaging QoS is implemented in the `TAO_Messaging` library. Thus, applications that use these features must link with this library. MPC projects for applications that use Messaging QoS can simply inherit from the `messaging` base project. For example, below is the MPC file for the Timeout test in `$TAO_ROOT/tests/Timeout` that uses the relative round-trip timeout policy:

```mpc
project(*Server): taoexe, portableserver {  
  Source_Files {
    test_i.cpp
    server.cpp
  }
}

project(*Client): messaging, taoexe, portableserver {
  requires += corba_messaging

  Source_Files {
    testC.cpp
    client.cpp
  }
}
```

For more information on MPC, see Chapter 4.

### 7.4 Bi-Directional GIOP

Bi-directional GIOP provides a solution to the problem of invoking callback operations on clients behind a firewall. Imagine you have a client application that resides on hosts that are inside firewalls. That client contacts a server outside of the firewall and provides a callback object for the server. When the server invokes upon the client callback object, the server attempts to open a new connection back to the client application. Unless the firewall at every
installation of the client is configured to allow access by the server, the
callback invocation fails. Configuring the firewall to allow the server to open
a callback connection may impose a significant installation cost and/or violate
site security policies.

Bi-directional GIOP solves this problem by allowing the callback invocation
to use the connection that already exists between the client and the server,
which is the connection that was used to transmit the client’s initial request to
the server. The server’s callback invocation doesn’t need to open a new
connection to the client; thus, a firewall does not block the callback.

Connection management restrictions imposed in GIOP versions 1.0 and 1.1
state that only clients can send requests, and only servers can respond to
requests. (By definition, an application or process that initiates a connection is
a client, and an application or process that accepts connections is a server.
Connection management, however, is orthogonal to the sending of requests
and replies.) This restriction can be overcome in GIOP v1.2 by specifying a
bi-directional policy of BOTH. This policy allows the server to invoke the
client’s callback operations, and the client to respond to these invocations, on
the same connection that the client established initially. The client and server
use only one connection instead of two.

The code in $TAO_ROOT/DevGuideExamples/BiDirectionalGIOP
provides an example of a bi-directional connection between a client and
server. The client creates a callback object and passes its object reference to
the server so that the server may invoke an operation on the callback object.
Bi-directional GIOP allows the server to invoke a callback operation on the
client without creating another connection to the client. There is also a test
case in $TAO_ROOT/tests/BiDirectional that behaves in a similar
fashion.

To create a bi-directional connection, both the client and server must specify a
BidirectionalPolicyValue of BOTH when creating their POAs. The default
policy is NORMAL.

### 7.4.1 Bi-Directional GIOP Example

The following example shows how to set the bi-directional GIOP policy on a
new POA:

```c
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```
// Get the RootPOA and its POAManager.
CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa = PortableServer::POA::_narrow(poa_obj.in());
PortableServer::POAManager_var poa_manager = root_poa->the_POAManager();

// Create policies for the child POA to be created.
CORBA::PolicyList policy_list;
policy_list.length(1);

CORBA::Any bi_dir_policy_as_any;
bi_dir_policy_as_any <<= BiDirPolicy::BOTH;
policy_list[0] =
    orb->create_policy(BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
    bi_dir_policy_as_any);

// Create a POA as a child of RootPOA with the above policies. This POA
// will receive requests on the same connection on which it sent the request.
PortableServer::POA_var child_poa =
    root_poa->create_POA("biDirPOA", poa_manager.in(), policy_list);

// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
    policy_list[i]->destroy();
}
policy_list.length(0);

// Activate both POAs.
poa_manager->activate();

// ... rest of application ...

7.4.2 Building Applications that use Bi-Directional GIOP

TAO's support of bi-directional GIOP is implemented in the TAO_BiDirGIOP
library. Thus, applications that use bi-directional GIOP features must link with
this library. MPC projects for applications that use bi-directional GIOP can
simply inherit from the bidir_giop base project. For example, below is the
MPC file for the bi-directional GIOP test in
$TAO_ROOT/tests/BiDirectional:

project(*Server): taoexe, portableserver, minimum_corba, bidir_giop {
    requires += exceptions
    Source_Files {
        test_i.cpp
        server.cpp
    }
}
Note: The minimum_corba base project, from which the above projects inherit, indicates that these projects will not build if minimum_corba=1.

For more information on MPC, see Chapter 4.

Note: The current implementation of TAO sets the BiDirPolicy at the ORB level rather than in each POA. Thus, every connection in that ORB will have the BiDirPolicy::BOTH value.

Note: Warning: There are security issues involved in using Bi-Directional GIOP. See the last paragraph of section 15.8, “Bi-Directional GIOP”, in the CORBA Core specification (OMG Document formal/04-03-12) for a complete description. See section 15.8.1.1, “IIOP/SSL Considerations,” for issues related to using Bi-Directional GIOP over IIOP/SSL.
CHAPTER 8

Asynchronous Method Handling

8.1 Introduction

Processing CORBA requests may require a long duration of activity resulting in the blocking of subsequent requests. This can reduce the server’s responsiveness as threads that might otherwise be used to service incoming requests must block while waiting for a response. Concurrency strategies such as thread-per-connection and thread-pool can be used in such cases to increase the system responsiveness, but these approaches may not scale well as the number of threads increase due to increased number of client connections or client requests. Asynchronous Method Handling (AMH) is a TAO specific feature that addresses this situation without requiring you to implement complicated concurrency strategies.

Note Although AMH is TAO specific, it has been submitted to the Object Management Group for possible inclusion into the CORBA specification.

AMH provides server implementers the means to have a request for an operation be handled by one thread, while having the response to that request
delivered by another thread. For example, a server that needs to invoke a remote operation in the course of handling a request can combine the use of AMH with Asynchronous Method Invocation (AMI), discussed in 7.2, to have an AMI callback handler complete the processing of the original invocation, and return the result to the caller. In this way, AMH greatly reduces the risk of a runaway stack that may arise from the use of concurrency and wait strategies such as the Leader-Follower strategy.

Clients of AMH based services are unaware of this responsibility hand-off. However, using AMH does impose some limitations:

- All requests must be passed as GIOP messages. No direct or through-POA collocation is possible with AMH. The use of AMH in an application must be anticipated at design time.

- AMH may be considered to violate certain aspects of the contract between servants, POAs, servant managers, and other server objects. This violation arises from the possibility that a thread calling a servant during an invocation may return control back to the POA before the actual operation is complete without throwing an exception.

Note  Code using AMH can be found in TAO itself. In particular, TAO’s Implementation Repository uses AMH to improve its performance when there are numerous clients trying to connect to it.

### 8.1.1 When AMH is Useful

Situations where AMH can be useful include:

- You have numerous client connections or client requests to a service that depends on systems or other services that may cause the serving thread to block. Scenarios that cause such blockage include media I/O, database access, reliance on long-running services, or computation intensive activities such as numerical analysis. A scenario where AMH is particularly useful is when it is used in conjunction with AMI for the middle-tier server in a three-tier architecture where the middle-tier server offloads requests to servers that can be time consuming to fulfill. For this scenario, neither the client nor the target server processing the request are aware of the AMH/AMI usage by the middle-tier server. Details on using AMH in combination with AMI can be found in 8.6.
• A very large number of clients are concurrent connected to your server. Empirical data from Design and Performance of Asynchronous Method Handling for CORBA shows that when the number of concurrent clients connected to a server gets large enough, when using standard concurrency strategies, the request throughput becomes unacceptably low. In this case, AMH becomes the only practical solution available.

• You wish to simplify concurrency support in your server code. Using AMH, it is possible to avoid multithreaded programming, which can be difficult to write and maintain.

• It is important that requests be processed in the order in which they are received. The special skeleton code generated for using AMH ensures that client requests are processed in the order received.

8.1.2 When AMH is not Useful

Some situations where AMH may not be applicable include the following:

• There will rarely be more than a medium load of concurrent clients. Additional empirical data provided in Design and Performance of Asynchronous Method Handling for CORBA shows that AMH is slightly less efficient than other concurrency models. However, you should weigh this slight reduction in efficiency to the possibly simpler server code that can be written when using AMH to support concurrent requests.

• You are using certain advanced CORBA features. Some advanced CORBA features assume that the thread that starts an invocation is the same one that finishes it. However, AMH breaks this assumption. Further discussion about using advanced CORBA features along with AMH can be found in 8.5.

• Your application can throw a number of exceptions. You must be careful when using AMH in an application that raises exceptions. AMH response handlers, the code responsible for sending a reply back to the client, are typically not invoked by a POA or a skeleton. Therefore, care must be taken to catch all exceptions in order to communicate them to the client. There is no framework that will assist in automating this task.
8.2 Participants in an AMH Servant

Server applications using AMH rely on the interaction of the following classes:

- A special skeleton class generated by the IDL compiler for supporting AMH.
- A Response Handler class also generated by the IDL compiler that takes the reply information and passes it to the client. An instance of this class is generated by the AMH skeleton.
- The AMH servant code you write.

AMH servants are derived from the AMH skeletons. The implementation methods of these skeletons differ from the ordinary server-side IDL to C++ mapping. Only in and inout style arguments are passed to implementation methods, and the method has a void return, regardless of the return type of the IDL operation. The response handler’s interface deals with the outputs for each operation.

The response handler is a local CORBA object. Its role is to gather any output (out and inout arguments, and return values) from an IDL operation and prepare a GIOP reply message for the client. A given response handler is only valid during a single operation. Once the reply is sent, references to the response handler should be discarded. For each IDL operation, the response handler has two methods, one for ordinary returns, and another for returning exceptions.

8.2.1 Simple Example
Consider the following IDL:

```idl
interface EchoTest
{
    string echo (in string message);
};
```

The following code fragments illustrate the participants in an AMH servant.

**AMH Skeleton**

```cpp
class AMH_EchoTestResponseHandler_ptr;
class POA_AMH_EchoTest
```
8.2 Participants in an AMH Servant

```cpp
{
public:
    virtual void echo (AMH_EchoTestResponseHandler_ptr rh, const char * message) = 0;
};

Response Handler
class AMH_EchoTestResponseHandler : public virtual CORBA::Object
{
public:
    virtual void echo (const char * return_value)
};

Servant
class AMH_EchoTest_i : public virtual POA_AMH_EchoTest, public virtual PortableServer::RefCountServantBase
{
public:
    virtual void echo (EchoTestResponseHandler_ptr rh, const char * message);
}

The following diagram illustrates how these participants interact to handle an invocation.

![Figure 8-1AMH Servant and Response Handler](image-url)
Asynchronous Method Handling

As shown in Figure 8-1, the ORB infrastructure processes the incoming request from the client and invokes the servant. The reply to the request is not sent until the application calls the appropriate response handler method. This can happen from the servant code, or the application may retain a reference to the response handler and send the reply at a later time.

8.3 Generating AMH Related Code

The TAO IDL compiler takes a single command line option, -GH, which triggers the generation of the AMH skeleton and response handler classes. The IDL compiler and its options are discussed at length in Chapter 5.

For each interface in the IDL file, the compiler generates an AMH skeleton and a response handler. The name for the AMH skeleton is similar to that of the ordinary skeleton, except that AMH is prepended to the interface name. Recall that CORBA compliant skeletons are named by prepending POA to the fully qualified interface name, including any modules. In general the form is POA[<modules>::]AMH_<interface>

For an interface declared as:

```idl
module DevGuide
{
  interface Messenger
  {
    ...
  };
};
```

The IDL compiler will generate POA_DevGuide::Messenger and POA_DevGuide::AMH_Messenger. If the interface were declared outside of any module, the generated names would be POA_Messenger and POA_AMH_Messenger.

The response handler generated by the IDL compiler for each interface is given a name derived from the interface name in this way: [<module>::]AMH_<interface>ResponseHandler. For example, the response handler for the interface above is DevGuide::AMH_MessengerResponseHandler.

Response handlers are reference counted local CORBA objects. As such, the response handler has a stub class, a var, ptr, and an out type related to it.
These are all necessary to allow a response handler instance to be created during the receipt of an invocation, then held until the invocation is complete. Response handler references are kept by the skeleton infrastructure until a reply or exception operation is invoked. It is the servant’s responsibility to either use the response handler or store its reference for future use. It is essential that either a reply or exception reply method be invoked before losing the reference to the response handler. Failure to do so will cause the client to hang.

8.4 An AMH Example Program

Now let us take a look at the Messenger example, extended to use an AMH based servant. The full source code for this example can be found at $TAO_ROOT/DevGuideExamples/AMH.

The IDL is nearly identical to that used in other code examples, except that here the Messenger interface is enclosed in a module so that the application of the naming convention can be seen.

```plaintext
module DevGuide
{
  interface Messenger
  {
    boolean send_message(in string user_name,
                          in string subject,
                          inout string message);
  };
};
```

The `send_message()` operation has data that flows in and out through its parameters, as well as returning a value. Also, even though no user exceptions are raised by the operation, it is still liable to raise system exceptions.

8.4.1 The Generated Stub Classes

When the IDL is compiled, a stub is created for the Messenger interface that is identical to the stub used for the non-AMH case. The same stub is used as a proxy for objects served by synchronous servants as well as asynchronous ones.
A stub for the response handler is also created. This stub is defined in the MessengerC.h and MessengerC.cpp files. Although they are of no interest to the clients, AMH-based servers will use these stubs. For the Messenger interface shown above, the response handler stub appears as follows:

```cpp
namespace DevGuide
{
    class AMH_MessengerResponseHandler : public virtual CORBA::Object
    {
    virtual void send_message (
        ::CORBA::Boolean return_value,
        const char * message
    ACE_ENV_ARG_DECL_WITH_DEFAULTS
    )
    ACE_THROW_SPEC ((
        CORBA::SystemException
    )) = 0;

    virtual void send_message_excep (
        ::DevGuide::AMH_MessengerExceptionHolder * holder
    ACE_ENV_ARG_DECL_WITH_DEFAULTS
    )
    ACE_THROW_SPEC ((
        CORBA::SystemException
    )) = 0;
    };
}
```

For each operation in the interface, two corresponding response handler member functions are created. The first, which has the same name as the operation, takes arguments for the operation’s return value, and any out (or inout) arguments. The second method generated for responding to an operation is used for sending exceptions back to the client. The name of this method is generated by appending _excep to the operation name. The exception reply method takes a single argument, a reference to an exception holder. The exception holder is a valuetype object that is capable of raising whatever exception it contains.

**Note** The use of an exception holder valuetype and inclusion of AMH stubs make the generated client-side files dependent on the valuetype library. Any application linked to these generated files must also be linked to the valuetype library. A way to minimize the impact of this dependency is to generate two
sets of stub definitions: one with AMH for use in server development, and one without for use on the client side.

The exception holder is a specialized value type object that is unique to a particular interface. The exception holder has a method for each operation that is responsible for raising whatever exception needs to be propagated back to the client. An example showing the use of the exception holder is shown in 8.6.1.

8.4.2 The AMH Servant

The TAO IDL compiler, when given the \(-\text{GH}\) option, generates an ordinary skeleton as well as an AMH skeleton for every interface. Using the example IDL shown above, the ordinary skeleton and AMH skeleton are as follows.

```cpp
namespace POA_DevGuide
{
  class Messenger : public virtual PortableServer::ServantBase
  {
    public:
      virtual CORBA::Boolean send_message (
        const char * user_name,
        const char * subject,
        char *& message
        ACE_ENV_ARG_DECL
      )
        ACE_THROW_SPEC ((CORBA::SystemException)) = 0;
  
  class AMH_Messenger : public virtual PortableServer::ServantBase
  {
    public:
      virtual void send_message (
        DevGuide::AMH_MessengerResponseHandler_ptr _tao_rh,
        const char * user_name,
        const char * subject,
        const char * message
        ACE_ENV_ARG_DECL
      )
        ACE_THROW_SPEC ((CORBA::SystemException)) = 0;
  
};
};
```
Note that the AMH skeleton’s definition of the `send_message()` method differs from that of the ordinary skeleton in two ways:

1. The leading argument is a reference to the response handler for this particular invocation. This response handler must be invoked by the servant for any operation to send a GIOP reply message to the client.
2. The remaining `send_message()` parameters map to the operation’s `in` and `inout` arguments. The return from `send_message()` is void, even though the IDL operation returns a string. The return value is passed to the appropriate method on the response handler, as are any out and out-bound `inout` values, if any are defined.

Now let’s take a look at a very simple implementation for the AMH version of `send_message()`. In this case, we are not really taking advantage of the benefit of AMH because we are directly invoking the response handler immediately from the servant.

```cpp
void AMH_Messenger_i::send_message(
    DevGuide::AMH_MessengerResponseHandler_ptr _tao_rh,
    const char * user_name,
    const char * subject,
    const char * message
)
ACE_THROW_SPEC((CORBA::SystemException))
{
    std::cout << "Message from: " << user_name << std::endl;
    std::cout << "Subject:      " << subject << std::endl;
    std::cout << "Message:      " << message << std::endl;
    CORBA::String_var inout_message_arg =
        CORBA::string_dup("Thanks for the message.");
    CORBA::Boolean result = 1;

    _tao_rh->send_message(result, inout_message_arg.inout());
}
```

The parameters passed to the response handler’s `send_message()` method are supplied using `in` parameter semantics. This means that the caller is still responsible for releasing memory that was used by any intermediate values that may be returned. So, for example, the response handler must duplicate the `in` parameters it receives.

This behavior exists due to an inversion of control that results from the asynchronous request processing. In an ordinary request/reply invocation, the servant has control only when the thread is in the implementation method,
thus control goes away when the method returns. Another way to look at this is that the lifespan of the invocation is only as long as the duration of the invocation method. This means that it is the responsibility of the caller of the method (the generated skeleton class) to clean up any allocated resources being passed back to the client.

By contrast, when using AMH it is possible for the lifespan of the invocation to exceed the duration of the invocation method. The example shown above happened to use the response handler right away, but it is perfectly valid to store a reference to the response handler, then invoke the appropriate method on it at a later time. For instance, the example code shown above alternatively could have spawned a thread to handle the response.

```cpp
#include <ace/Thread.h>
void
AMH_Messenger_i::send_message (
    DevGuide::AMH_MessengerResponseHandler_ptr _tao_rh,
    const char * user_name,
    const char * subject,
    const char * message
) ACE_THROW_SPEC((CORBA::SystemException))
{
    std::cout << "Message from: " << user_name << std::endl;
    std::cout << "Subject: " << subject << std::endl;
    std::cout << "Message: " << message << std::endl;
    DevGuide::AMH_MessengerResponseHandler_ptr dup_rh =
        DevGuide::AMH_MessengerResponseHandler::_duplicate (_tao_rh)
    ACE_Thread::spawn (send_message_reply, dup_rh);
};
```

The response handler reference must be managed, and sending the response does not end the method. The response handler is a reference counted local CORBA object. An instance of the response handler is created in the skeleton just prior to calling the implementation method, and the skeleton releases its reference to the response handler when the implementation method returns. This is why, in the implementation of `send_message()` above, the servant duplicates the reference to the response handler before returning.

```cpp
void send_message_reply (void * arg)
{
    DevGuide::AMH_MessengerResponseHandler_ptr rh =
        (DevGuide::AMH_MessengerResponseHandler_ptr)arg
    CORBA::String_var inout_message_arg =
```
Asynchronous Method Handling

```c++
CORBA::string_dup("Thanks for the message.");
CORBA::Boolean result = 1;
ACE_OS::sleep (5);

rh->send_message(result,inout_message_arg.in());
CORBA::release (rh);
```

In this case, the response handler is passed to a thread function that waits a few seconds before proceeding to send the reply. Here, the invocation spans the life of both the implementing method called by the skeleton, as well as the second thread function. The first method returns to the caller immediately after spawning the thread, but the client does not receive a reply until after the thread function completes. The thread function must release a reference to the response handler to offset the reference duplication done in `AMH_Messenger_i::send_message()`.

Invoking the response handler CORBA object results in the sending a response back to the original client without having any effect on any of the values passed to it. Therefore, any allocation of storage, such as the string shown above, is still available for reuse (or leaking if not managed properly).

### 8.4.3 AMH and Oneway Invocations

Operations that have no out or inout arguments and a void return type will still have a response handler that must be called. This is also true for oneway operations. Oneways may be invoked by a client that has set the `SYNC_WITH_TARGET` synchronization scope policy, which requires the server to send a GIOP response when the operation completes. See 7.3.4 for more details on this policy.

### 8.4.4 Throwing Exceptions

Exceptions are sent back to clients using a special form of the GIOP reply message. The message header contains a flag indicating that the reply contains an exception and the message data contains the marshaled exception. Because of the inversion of control mentioned above, we cannot simply throw an exception and expect it to be propagated back to the original client. Exceptions are thrown by the use of specialized methods on the response handler. Along with each method for returning values (even if void) from an interface’s attributes and operations, the response handler also has a method...
for each of these to raise exceptions. These methods are named by appending \_excep to the name of the operation to which it is related. Thus, the DevGuide::AMH_MessengerResponseHandler::send_message() method shown above is accompanied by send_message_excep() for raising exceptions.

There is a distinct exception method for each operation. The data supplied to the method is a reference to a value type object that contains the exception to be thrown. This exception holder object is similar to the exception holder defined by the CORBA Messaging specification. The difference being that in AMI, the framework calls the \_excep method you implement in your callback handler, whereas with AMH, you call the \_excep method supplied to you by the framework.

The following example code shows how to throw an exception via the response handler. To make this example a little clearer, a new interface is defined with an operation that raises a user exception.

```cpp
module DevGuide
{
    exception demo {}; 
    interface Asynch_Except_Demo 
    {
        boolean trigger () raises (demo);
    }
};

This could be implemented by an AMH servant method such as this one:

```cpp
void 
Excep_Example_i::trigger (DevGuide::AMH_Asynch_Except_DemoResponseHandler_ptr 
_tao_rh) 
{
    DevGuide::demo *d = new DevGuide::demo;
    DevGuide::AMH_Asynch_Except_DemoExceptionHolder ex_holder(d);
    _tao_rh->trigger_excep (&ex_holder);
    std::cout << "Done sending exception." << std::endl;
}
```

You will see there is something peculiar about this code example. The exception instance, d, is a pointer to an exception allocated on the heap. This code does not free the exception object after invoking the exception sender. This is because the exception is given to the exception holder, which takes ownership of the exception.
The AMH exception holder is initialized by supplying a pointer to an exception instance allocated on the heap. This requirement is a consequence of the mechanism used to propagate the exception back to the caller. This is done by the response handler invoking the operation-specific raise method of the exception holder, which in turn guards the pointer to the exception with an auto-pointer, then calls a method on the exception that causes it to throw itself. Afterward, the auto-pointer releases the holder’s reference to the exception, causing it to be deleted.

Remember that invoking the appropriate response method was semantically identical to making an invocation with `in` parameters. The exception methods violate this notion because these methods require the exception to be allocated on the heap, and consume it as a side effect of sending the exception.

This behavior results from the mechanism used to actually form and transmit the GIOP message containing the exception. This code relies on the exception holder to throw the exception, then uses existing skeleton helper methods to generate the GIOP method. By throwing the exception, the exception holder extracts the exception instance from the exception holder, assigns it to an auto pointer, then throws the exception.

### 8.4.5 AMH And The Server Main

A process hosting objects served by asynchronous servants is only different from one hosting synchronous objects in that it has a choice as to which type of servant it wishes to attach to an activated object. The server must still initialize an ORB, obtain a POA, and use that POA to associate the servant (asynchronous or synchronous) with an object.

### 8.4.6 AMH and the Client

As with the main function of the server, clients are unaware of the synchrony of the servant behind any objects it uses. Currently, TAO 1.4a imposes a side-effect on clients of AMH enabled services. The AMH response handler and exception holder classes are defined in the generated stub header file. The exception holder is a value type, as is the exception holder in an AMI callback object. This creates a dependency between the client application and the TAO value type library. This dependency has existed since prior to TAO 1.4a, but the value type support was directly part of TAO in earlier versions.
There is a plan to separate the AMH generated stubs from the ordinary client code, and this will likely be added to TAO 1.4a as a patch when it is available.

A second side-effect is manifest in MPC files related to applications that directly build the IDL for an AMH based server. As is the case with the AMH example code provided in $TAO_ROOT/DevGuideExamples/AMH, the client subproject must be dependant on the AMH base project, amh.mpb. This ensures that if this project is used to compile the IDL, the proper AMH elements will be generated. For example, the definition for an MPC project that inherits from the AMH base project could look like:

```cpp
project(*Client): ..., amh
{
...
}
```

## 8.5 AMH and Advanced CORBA Features

CORBA servers frequently have many details to manage in addition to running implementation code. Examples include:

- An application may make use of servant managers to control the deployment of servant instances.
- Applications may make use of certain CORBA Current objects to gain access to information that is specific to a particular invocation context.
- Applications may be written in such a way that a servant is collocated with a client, and may wish to use strategies such as direct collocation for performance optimization.

Many of these advanced features of CORBA assume that the thread that starts an invocation is the same thread that finishes it. AMH makes it possible to invalidate this assumption. Therefore, care must be taken to ensure that any invocation-specific context data is separately managed so that any subsequent thread that participates in an invocation is able to access this information. The following subsections outline some of the obvious situations where such context dependent information may be needed, although it is not a complete list.
8.5.1 Portable Interceptors
Portable Interceptors are specialized client or server objects that are used to process inbound or outbound messages, very close to the transport layer. Interceptors are used to initialize the context within any CORBA Current objects used by the server, and they may be used to process service contexts, which contain meta-information attached to a message. For example, this meta-information might include security information such as authorization tokens or credentials, or may include transaction tracking information.

When using any service that relies on interceptors, you should be very careful when AMH is also used. For example, TAO’s RT Scheduling Service assumes that all responses will be sent by the same thread as that on which the original request was received.

If you are implementing your own interceptor to manage meta-information, such as request auditing or other activities, and you use AMH, be sure to write your interceptor in such a way as to avoid relying on any information that is specific to the thread receiving a request or sending a reply.

8.5.2 Servant Locators
Servant managers work with the POA to supply servant instances on demand. As discussed in Advanced CORBA Programming with C++, 11.7.3, there are two kinds of servant managers: activators and locators. Activators provide a servant to the POA that is retained in the POA’s active object map, and remains associated with a given object until that object is deactivated or the POA itself shuts down. Locators are servant managers that provide servants only for the duration of a single invocation. Because of the limited lifespan of servants when using locators, care must be taken using AMH when servants are managed using locators.

Whether the servant is held by the POA for the duration of a single invocation or many, it is handed back to the servant manager for clean up. With AMH, this may occur before the response handler has sent a reply message. If you are using AMH and a servant locator then you must ensure that any reference to the original servant is not used, as the servant locator may have destroyed the servant. Similarly, you may have implemented the servant locator to manage a pool of servant objects. While the servant instance may still be valid, you must be aware that any state information that you update while handling an invocation may be modified if the invocation is handed off to another thread for completion.
8.5.3 Invocation Related CORBA::Current Objects

CORBA Current objects are locality constrained interfaces, derived from the empty CORBA::Current interface, that give application code access to information that is specific to the current thread of control. This access is provided through an object reference supplied by the ORB. Some current objects, such as PortableServer::Current and PortableInterceptor::Current, are specific to the current invocation. Because this invocation information is thread-specific, it is not sufficient to hand a current object to whatever thread will complete an AMH invocation. The initial servant thread must extract any information from the current and hand that data separately to the invocation completing thread. If a completion handler needs invocation-specific information from the current, such as the object ID from a PortableServer::Current, the information must be accessed in the original servant method and handed off to the completing thread along with the response handler.

Since current objects are context aware, the completing thread cannot simply use a reference to the invocation related current. Any invocation on the current by that thread would result in a NO_CONTEXT exception being raised. From the point of view of the current, the second thread is outside of the context of an invocation.

8.5.4 Reference Counted Servants

Servants may be reference counted in order to avoid memory leaks when objects associated with the servants are destroyed. A servant’s reference count is incremented during an invocation and decremented again when the invocation completes. This way, an invocation that deactivates an object, such as one calling POA::deactivate_object(), will not cause a crash when the POA removes its reference to the servant.

When an AMH servant method returns, regardless of the disposition of any pending reply, the skeleton code will treat this as a completion of the invocation and will decrement the reference count on the servant. This means that if your AMH servant uses a companion object to complete an invocation, the companion object should either have no association with the original servant, or must manage the reference count of that servant. Managing the reference count may be done explicitly by calling _add_ref() and _remove_ref() on the servant, or implicitly by holding the reference to the servant in a PortableServer::ServantBase_var.
8.5.5  Collocation

Collocation occurs when the servant for an object is in the same address space as the client making an invocation on that object. There are two forms of collocation, thru-POA and direct. Collocated invocations that go through the POA have an opportunity to also go through interceptors and are subject to the control imposed by the POA such as being rejected because the POA is in the discarding state. Direct invocations are forwarded straight from the stub method to the servant method. In both cases, the caller’s thread is actually used to perform the invocation.

Since an AMH servant’s implementation of an operation has a different signature than what is expected based on the IDL definition of the operation, it does not match what is expected by the collocated stub. Further, AMH makes it possible or likely that the invocation is not complete when the servant method returns. There is currently no mechanism in TAO’s collocated stubs to enable the calling thread to wait until some other thread invokes the response handler and provides results to the caller. Therefore, there is no support in TAO for combining collocation and AMH.

Given that the desire to use AMH is often the result of bottlenecks in the servant, we might find that the efficiency gained by using collocated calls that avoid marshaling would be minimal anyway. Therefore, it is reasonable to explicitly disable collocated calls in an application that is implemented using AMH based servants. Keep in mind that such control is imposed at IDL compilation time. Be sure to isolate the definitions of interfaces that will be implemented using AMH from those for which you wish to support collocated access.

---

**Note**  
*Future versions of TAO may support collocated invocation of AMH servants.*

8.6  Combining AMH with AMI

CORBA servers are often used in multi-tier applications, with middle layers serving as concentrators or gateways. The middle layer may process requests by turning around and sending invocations to other servers to complete. Consider the following IDL interfaces:

```
// file: middle.idl
```
interface Middle
{
    string get_the_answer (in string question);
};

// file: inner.idl
interface Inner
{
    string answer (in string question);
};

A client might invoke \texttt{Middle::get\_the\_answer()}, which in turn invokes \texttt{Inner::answer()}, waits for the response, then replies back to the client with the answer. Imagine that invoking \texttt{Inner::answer()} takes a long time, and that the client load is variable. Sometimes there may be two or three client requests pending, while at other times there may be hundreds. The thread waiting for a reply is essentially a wasted resource. As the client load increases, if there are insufficient threads available to handle the load, then clients may not be able to have their requests processed within their time constraints.

Traditionally, to avoid making a thread a wasted resource, the server implementing the \texttt{Middle} interface would process a request invoking \texttt{get\_the\_answer()} in a separate thread using some threading strategy. Before this thread returns, it in turn invokes \texttt{Inner::answer()} and waits for the result. While waiting, this thread may be blocked so that it cannot handle other requests. If TAO’s thread pool strategy is being used, and the thread is part of a thread pool, then it is at risk of being “borrowed” to process another incoming request. This is illustrated in Figure 8-2.

\begin{note}
Further discussion about TAO’s threading models and wait strategies can be found in Chapter 17.
\end{note}
Asynchronous Method Handling

In addition, using TAO’s thread pool strategy raises the possibility of recursive requests (also called “nested upcalls”), wherein a thread waiting for the Inner::answer() response may be required to handle another incoming client request. Of course, the problem could be resolved by adding more threads, but this does not scale well as the number of client requests increases. This is because context switching between many threads may overwhelm the system.

Figure 8-2 Middle-tier Server Without AMH and AMI
The best solution to this problem is to implement the Middle server using AMH to handle the incoming `get_the_answer()` requests and AMI to invoke `Inner::answer()`, as shown in Figure 8-3.

![Diagram](image)

**Figure 8-3 Middle-tier Server Using AMI and AMH together**

### 8.6.1 AMH/AMI Example

An example implementation of the Middle server using AMH and AMI together is given here. The full code for this example is in the directory `$TAO_ROOT/DevGuideExamples/AMH_AMI`.

#### 8.6.1.1 AMI Callback Handler

To realize the benefits of AMH/AMI, an AMI callback handler in the middle tier must be supplied to manage asynchronous replies from the Inner server. Since AMI callback handlers are CORBA servants we must supply an implementation for the handler. `Inner_callback_i` is the AMI callback handler invoked when the reply to `Inner::answer()` is received. The role of this callback handler is to forward the answer back to the original caller. Therefore, the callback servant must be initialized with the correct response handler shown here:
Asynchronous Method Handling

class Inner_callback_i : public virtual POA_AMI_InnerHandler,
        public virtual PortableServer::RefCountServantBase
{
public:
    Inner_callback_i (PortableServer::POA_ptr poa,
              AMH_MiddleResponseHandler_ptr _tao_rh);
    virtual void answer (const char * ami_return_val);
    throw(CORBA::SystemException);
    virtual void answer_excep (AMI_InnerExceptionHolder * excep_holder)
    throw(CORBA::SystemException);

private:
    PortableServer::POA_var poa_
    AMH_MiddleResponseHandler_var response_handler_
};

As you will see below, we will make sure the AMI reply handler is initialized with the AMH response handler. The callback handler reference must be supplied for each sendc_call, in order to allow the ORB’s dispatcher to deliver the reply to the appropriate handler.

As is the case for all AMI callback handlers, our callback handler has two methods that must be supplied for each operation in the interface. One for dealing with ordinary replies, another for dealing with exceptions. In this example, we have a callback handler class, Inner_callback_i, with methods answer() and answer_excep().

Note Because we are using AMI, the Inner server does not need to change to support AMI-based asynchronous replies.

Consider now the handling of ordinary replies:

void
Inner_callback_i::answer (const char * ami_return_val)
    throw (CORBA::SystemException)
{
    this->response_handler_->_get_the_answer (ami_return_val);
    PortableServer::ObjectId_var oid = this->poa_->_servant_to_id(this);
    this->poa_->_deactivate_object (oid.in());
}

Handling return values is straightforward. The return value and any out/inout parameters are supplied as in parameters to the callback handler.
It simply passes those values on to the AMH response handler, again as in parameters, which forwards the response to the original client. After that, this servant has done its job, so it deactivates itself.

Now consider the handling of exceptions. As shown in section 8.4.4, the AMH exception holder is initialized with a local copy of the exception extracted from the AMI exception holder. Combining this behavior with the AMI exception callback interface requires that we explicitly duplicate the exception in order to pass it on to the client. This may be done by having the AMI exception holder throw the exception, which we then catch, duplicate, and pass to the client via the AMH response handler. This technique is shown here.

```c++
void Inner_callback_i::answer_excep (AMI_InnerExceptionHolder* excep_holder)
throw (CORBA::SystemException)
{
    try {
        excep_holder->raise_answer();
    }
    catch (CORBA::Exception& ex) {
        CORBA::Exception* local_ex = ex._tao_duplicate();
        AMH_MiddleExceptionHolder amh_excep_holder (local_ex);
        this->response_handler_->get_the_answer_excep (&amh_excep_holder);
    }
    catch (...) {
        CORBA::Exception* unknown_ex = new CORBA::UNKNOWN;
        AMH_MiddleExceptionHolder amh_excep_holder (unknown_ex);
        this->response_handler_->get_the_answer_excep (&amh_excep_holder);
    }

    std::cout << "inner_callback_i deactivating self" << std::endl;
    PortableServer::ObjectId_var oid = this->poa_->servant_to_id(this);
    this->poa_->deactivate_object (oid.in());
}
```

The method for handling any exceptions returned by the call to *answer()* is a little more involved. As discussed in 7.2.2, AMI delivers exceptions using value types also known as exception holders. To deliver the exception to the client, we must get the AMH-based exception holder and place it in the AMI-based exception holder. As with the AMI callback handlers and AMH response handlers, exception holders are generated for each interface.
8.6.1.2 AMH Servant

Consider now the interface to servant for the middle-tier servant:

```cpp
#include "middleS.h"
#include "innerC.h"

class Asynch_Middle_i :
    public virtual AMH_POA_Middle,
    public virtual PortableServer::RefCountServantBase
{
    public:
        Asynch_Middle_i (PortableServer::POA_ptr poa, Inner_ptr inner);
        virtual void get_the_answer (AMH_MiddleResponseHandler_ptr _tao_rh,
            const char * question)
            throw (CORBA::SystemException);
    private:
        PortableServer::POA_var poa_
        Inner_var inner_;
};
```

The middle servant is the only piece that needs to be AMH aware, therefore it is the only class to derive from a POA_AMH_* base class.

`Asynch_Middle_i::get_the_answer()` is passed an AMH response handler, which is used to initialize an instance of an AMI reply handler, as shown below:

```cpp
void
Asynch_Middle_i::get_the_answer (AMH_MiddleResponseHandler_ptr _tao_rh,
    const char * question)
{
    PortableServer::ServantBase_var servant =
        new Inner_callback_i (this->poa_.in(),_tao_rh.in());

    PortableServer::ObjectId_var objid =
        this->poa_)->activate_object (servant.in());
    CORBA::Object_var obj = this->poa_)->id_to_reference(objid.in());
    AMI_InnerHandler_var cb = AMI_InnerHandler::_narrow(obj.in());
    this->inner_)->sendc_answer (cb.in(),question);
}
```

The call to `Inner::sendc_answer()` sends the invocation request message to the Inner server and returns immediately.

In general, after this point the servant thread is free to handle any other incoming messages, whether they are new requests from clients, or AMI replies from the Inner server. The AMI reply handler will be invoked on some
thread when the Inner server sends a reply message from the answer() invocation. It will then immediately invoke the get_the_answer() method of the AMH response handler, which sends a reply back to the originating client.
CHAPTER 9

Real-Time CORBA

9.1 Introduction

In 1999, the OMG introduced the Real-Time CORBA specification (ptc/99-06-02) (RT CORBA 1.0) to provide CORBA developers with policies and mechanisms for controlling allocation of system resources and improving the predictability of system execution. The RT CORBA 1.0 specification was originally defined as a set of extensions to the CORBA core and the CORBA Messaging specifications. In August 2002, the OMG published a minor revision of the RT CORBA specification, RT CORBA 1.1 (formal/02-08-02). RT-CORBA 2.0 (formal/03-11-01), released in November 2003, introduced the dynamic scheduling model. Finally, RT-CORBA 1.2 (formal/05-01-04) integrated the dynamic scheduling details with the existing static scheduling model. Despite the unusual version numbering sequencing, RT-CORBA 1.2, released in January of 2005, is the most current version of the RT CORBA specification.
9.1.1 Road Map

In this chapter, we explore the topic of real-time CORBA from the perspective of an application with real-time predictability requirements, as well as from the perspective of the features available in TAO’s implementation. While the chapter is designed to be taken as a whole, you may find benefit to reading certain sections independently.

If you want to learn more about...

- The motivation for and scope of RT CORBA, see 9.2, “Real-Time CORBA Overview.”
- The new modules and interfaces introduced by RT CORBA, see 9.3, “Real-Time CORBA Architecture.”
- The latest dynamic scheduling features of RT CORBA, see 9.4, “Dynamic Scheduling.”
- Building and configuring applications that use TAO’s implementation of RT CORBA, see 9.5, “TAO’s Implementation of Real-Time CORBA.” This section also discusses TAO’s extensions to RT CORBA.
- Sample code that uses TAO’s RT CORBA features, see 9.6, “Client-Propagated Priority Model,” 9.7, “Server-Declared Priority Model,” and 9.8, “Using the RTScheduling::Current.” These sections present examples of client and server application code that makes use of various RT CORBA features and priority models. In addition, 9.9, “Real-Time CORBA Examples,” lists further examples and tests in TAO that use several of the RT CORBA features discussed in this chapter.

9.2 Real-Time CORBA Overview

The standard CORBA specification has historically done a very good job of supporting the requirements of distributed object-oriented systems, such as location transparency, programming language and operating system independence, separation of interface from implementation, and interoperability in heterogeneous environments.

However, in real-time systems, the timeliness of a system is as important as its functional requirements. That is, success is determined not only by logical correctness, but also by the time required to reach a correct solution or complete a task. A correct result that is reached outside the predetermined
time interval is still considered a failure. Such systems must be predictable and deterministic.

The applicability of CORBA to real-time systems has been limited, due to CORBA’s lack of standard mechanisms for specifying and enforcing Quality of Service (QoS) across distributed objects and supporting real-time programming techniques.

The goal of the RT CORBA specification is to address the shortcomings of CORBA for distributed real-time systems without sacrificing the spirit of CORBA and without placing a burden on developers of non-real-time systems. RT CORBA adds QoS control to standard CORBA with the goal of improving application predictability. RT CORBA achieves this by bounding priority inversions and managing resources end-to-end.

Specifically, RT CORBA provides policies and mechanisms for resource configuration and control in the following areas:

- **Processor Resources:**
  - RT CORBA defines *portable priorities* and a mechanism for mapping them to native operating system priorities.
  - RT CORBA enables *end-to-end priority propagation* via standard priority models and mechanisms so that clients and servers can specify request-priority propagation semantics.
  - RT CORBA adds *thread pools* and mechanisms for servers to allocate, partition, and manage thread characteristics.
  - RT CORBA defines *standard synchronizers* for coordinating contention for system resources in a consistent fashion.
  - RT CORBA defines *distributable threads* and *.schedulers* for managing static or dynamic scheduling.

- **Communication Resources:**
  - RT CORBA adds *protocol properties* to enable selection and configuration of protocols by clients and servers.
  - RT CORBA enables mechanisms for *explicit binding* to establish and manage connections between clients and servers.

- **Memory Resources:**
- RT CORBA enables *request buffering* by servers when all available threads are currently servicing requests.

Figure 9-1 shows how various RT CORBA policies and mechanisms (in *italics*) relate to the standard CORBA architecture.

*Figure 9-1 Real-Time CORBA Policies and Mechanisms*

To achieve its goals with regard to the above QoS policies, the RT CORBA specification leverages the QoS policy framework defined in the CORBA
9.3 Real-Time CORBA Architecture

Messaging specification. See Chapter 7 for more information on CORBA Messaging.

For example, using the QoS policy framework, a client can override default policy settings at the ORB, thread, or object reference level to affect qualities such as request priority, message delivery, and request/reply timeouts. Likewise, servers can use QoS policies with the Portable Object Adapter’s create_POA() operation to affect server-side qualities such as request queuing and the creation and management of thread pools.

The remainder of this chapter describes the specific QoS policies addressed by the RT CORBA specification and how they are supported by TAO. We extend the Messenger example from previous chapters to show how to use TAO’s implementation of the RT CORBA specification to address the QoS requirements of real-time applications.


9.3 Real-Time CORBA Architecture

The RT CORBA specification extends the standard CORBA specification with the addition of several new modules and interfaces to achieve end-to-end predictability and control over the management of resources. Developers of non-real-time CORBA applications need not be burdened by these extensions. RT CORBA extensions to the standard CORBA architecture include:

- Real-time ORB (RTCORBA::RTORB)
- Real-time POA (RTPortableServer::POA)
- Real-time CORBA priority (RTCORBA::Priority)
- Real-time Current (RTCORBA::Current)
- Real-time mutex (RTCORBA::Mutex)
- Thread pools (RTCORBA::ThreadPoolId)
- Thread pool lanes (RTCORBA::ThreadPoolLane)
Figure 9-2 shows how key entities defined by the RT CORBA extensions relate to the standard CORBA architecture.

Figure 9-2 also shows elements of Dynamic Scheduling, an extension to the real-time CORBA specification. Dynamic Scheduling, its components, features, and services are discussed at length in 9.4.

9.3.1 Real-Time CORBA Modules
The RT CORBA specification introduces additional IDL modules, RTCORBA RTPortableServer, and RTScheduling, which contain definitions of RT CORBA interfaces and types.

- The RTCORBA module contains definitions for portable priorities, priority mapping, thread pools, real-time policies, and protocol properties. It also contains definitions for real-time Current, Mutex, and ORB interfaces. Entities defined in the RTCORBA module are used by both clients and servers.
9.3 Real-Time CORBA Architecture

- The RTPortableServer module contains the definition of the real-time Portable Object Adapter (POA) interface for use by servers.
- The RTScheduling module defines the components used to support dynamic and static scheduling and distributable threads. The RTScheduling::Current may be used in both client and server processes.

**Note** The interface definition for module RTCORBA is quite large. We will be examining portions of it throughout this chapter. The definition is included in the TAO source distribution in $TAO_ROOT/tao/RTCORBA/RTCORBA.pidl. Likewise, the definition of RTPortableServer is found in $TAO_ROOT/tao/RTPortableServer/RTPortableServer.pidl and the definition of RTScheduling is found in $TAO_ROOT/tao/RTScheduling/RTScheduler.pidl.

9.3.2 The Real-Time ORB

The RT CORBA specification introduces an interface for a real-time ORB, RTCORBA::RTORB. The RT ORB is a local interface used to create resources necessary to manage real-time applications. The RTORB interface does not inherit from CORBA::ORB, rather it is simply a helper object that applications use to create and manage instances of various RT CORBA types, such as mutexes, thread pools, and policies.

TAO’s RT CORBA library supplies the implementation of RTORB. The RT CORBA library uses specialized ORBInitializers, as defined in the Portable Intercepter specification, to initialize the RTORB. Each ORB has an RTORB instance. Applications obtain a reference to the RTORB by calling resolve_initial_references("RTORB") on the ORB.

Rather than showing the entire RTORB interface in one place, the various operations are introduced individually in subsequent sections as we describe how to use them to create and manage certain RT CORBA objects.

The RTCORBA::RTORB is initialized when its associated CORBA::ORB is initialized during CORBA::ORB_init(). The RT CORBA specification defines a new ORB initialization parameter, ORBPriorityRange, used to constrain the range of CORBA priorities the RTORB may use. However, this option is not currently supported by TAO, meaning that any CORBA priority...
value may be used by any RTORB. CORBA Priority values are discussed in 9.3.4.

Here we show an example that obtains the RTORB from the ORB. For the sake of clarity, we have omitted error-handling code:

```c
#include <tao/corba.h>
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[]) {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get the RTORB.
    CORBA::Object_var obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rt_orb = RTCORBA::RTORB::_narrow (obj.in());

    // Use the RTORB to access RT CORBA features (e.g., create_threadpool() )
}
```

### 9.3.3 The Real-Time POA

The RT CORBA specification introduces the real-time POA interface, `RTPortableServer::POA`, which specializes `PortableServer::POA`. As shown below, the real-time POA adds new reference creation operations that accept a priority as an additional parameter.

```plaintext
module RTPortableServer
{
    local interface POA : PortableServer::POA
    {
        Object create_reference_with_priority (in CORBA::RepositoryId intf,
                                             in RTCORBA::Priority priority)
        raises (WrongPolicy);

        Object create_reference_with_id_and_priority
            (in PortableServer::ObjectId oid,
             in CORBA::RepositoryId intf,
             in RTCORBA::Priority priority)
        raises (WrongPolicy);

        PortableServer::ObjectId activate_object_with_priority
            (in PortableServer::Servant p_servant,
             in RTCORBA::Priority priority)
        raises (ServantAlreadyActive, WrongPolicy);
    }
}
```
void activate_object_with_id_and_priority (in PortableServer::ObjectId oid,
in PortableServer::Servant p_servant,
in RTCORBA::Priority priority)
    raises ( ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy );
};

When an application links to the RTPortableServer library, all POA references are implemented by the real-time POA. Thus an application may create a POA using the ordinary method of invoking PortableServer::POA::create_POA(), then narrow the newly-created POA reference to RTPortableServer::POA. The following example shows this technique. Once again, error checking has been omitted for the sake of clarity:

#include <tao/corba.h>
#include <tao/RTCORBA/RT_POA.h>

int main (int argc, char* argv[])
{
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get the RootPOA.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());

    // Create a child POA.
    CORBA::PolicyList policies;
policies.length(2);
policies[0] = poa->create_lifespan_policy (PortableServer::PERSISTENT);
policies[1] = poa->create_id_assignment_policy (PortableServer::USER_ID);
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    PortableServer::POA_var child_poa = poa->create_POA ("Child POA", mgr.in(), policies);

    // Use the new POA as a RT POA.
    RTPortableServer::POA_var rt_poa =
        RTPortableServer::POA::_narrow (child_poa.in());
}
9.3.4 Real-Time Priority Mapping

A Real-Time Operating System (RTOS) must support the concept of discrete thread priority to adequately leverage benefits of RT CORBA. Thread priorities are usually represented as a range of integer values and a direction of precedence. A thread’s priority is used to determine its execution eligibility, with threads having higher precedence being eligible for execution ahead of threads with lower precedence. RT CORBA refers to an operating system’s representation of priority as native priority. Native priorities are not used directly by RT CORBA, but are represented in IDL to provide a mechanism for allowing native code to interact with RT CORBA. The native priority interface is:

```plaintext
module RTCORBA {
    typedef short NativePriority;
    // ...
}
```

A short integer has the range -32768 to 32767, however only a subset of this range will be valid in any particular operating system.

To allow all objects participating in a distributed real-time application to have a consistent notion of thread priority, RT CORBA supplies a second type to represent portable priority:

```plaintext
module RTCORBA {
    typedef short Priority;
    const Priority minPriority = 0;
    const Priority maxPriority = 32767;
    // ...
};
```

The type `Priority` is used when referring to “portable” priority values. Although it is a signed short integer, an RT CORBA priority value is always positive, its range being constrained by the constants `minPriority` and `maxPriority`. With CORBA priorities, higher values take precedence over lower values.

Conversion between native priority and RT CORBA priority is achieved through mapping functions. The RT CORBA specification uses the following declaration to represent the mapping between priority types:

```plaintext
module RTCORBA {

```
The specification recognizes that this mapping behavior is frequently invoked. To minimize performance impact, the declaration uses a native type rather than an interface. Use of an interface would necessitate the use of a CORBA object, and require obtaining a reference to the same. On the other hand, native types are specified directly in the target language binding. The C++ binding for `PriorityMapping` is a class in the RTCORBA namespace.

```cpp
namespace RTCORBA {
    class PriorityMapping {
        public:
            virtual CORBA::Boolean to_native (RTCORBA::Priority corba_priority,
                RTCORBA::NativePriority& native_priority ) {
            }
            virtual CORBA::Boolean to_CORBA (RTCORBA::NativePriority native_priority,
                RTCORBA::Priority& corba_priority ) {
        }
    };
}
```

The methods `to_native()` and `to_CORBA()` may be called several times by an ORB during an invocation. To provide the greatest possible efficiency these methods do not throw exceptions, not even CORBA System exceptions. However these functions will return `FALSE` if the input value is outside the allowed range for that type. For both of these functions, the first argument has the semantics of an in, supplying input, the second argument being an out for receiving the converted results.

An RT ORB conforming to the specification will make use of these mapping functions throughout the course of an invocation. If the call to either mapping function returns `FALSE`, the ORB is required to stop processing the invocation and throw a `DATA_CONVERSION` system exception to the invoking application. Note that this exception may not be propagated if the error is the result of a oneway operation.

### 9.3.5 The Real-Time Current

The `CORBA::Current` interface serves as a common base for interfaces providing context-specific information during upcalls. Applications use the `RTCORBA::Current` to determine the priority of the current invocation.
Real-Time CORBA

```c
local interface Current : CORBA::Current
{
  attribute Priority the_priority;
};
```

The real-time Current is obtained by narrowing the reference to the object returned by calling `resolve_initial_references("RTCurrent")`. The `RTCORBA::Priority` value obtained from the Current may be mapped to a native priority by using the `PriorityMapping` object, as discussed in section 9.3.4.

### 9.3.6 The Real-Time Mutex

Real-time CORBA defines the local `Mutex` object to present a portable interface for controlling access to sections of code. RT CORBA mutexes are created by the RTORB. The RT CORBA Mutexes are local objects, thus references to the Mutexes are not allowed to cross process boundaries.

The interface definition for Mutex is in the `RTCORBA` module. Shown here is the Mutex definition, along with operations on the RTORB which manage the life cycle of a CORBA Mutex.

```c
module RTCORBA
{
  // Mutex.
  local interface Mutex
  {
    void lock ();
    void unlock ();
    boolean try_lock (in TimeBase::TimeT max_wait);
    // if max_wait = 0 then return immediately
  };
};
```

A real-time CORBA Mutex is functionally similar to a common mutual exclusion lock, which is used to ensure that only one thread has access to critical sections of code at a time. A mutex has two states, locked and unlocked. A mutex starts out in the unlocked state. The mutex operations are:

- `lock()` sets the mutex state to locked, when called on an unlocked mutex. If the mutex is already locked, then `lock()` blocks until the owning thread calls `unlock()`. A mutex is not recursive, therefore if a thread attempts to call `lock()` twice, it will deadlock.
• **try_lock()** attempts to set the state of a mutex to locked. It returns `TRUE` if it successfully locks the mutex, or `FALSE` if the mutex cannot be locked within the `max_wait` time period. If zero is passed as the `max_wait` time, `try_lock()` immediately returns `FALSE` if the mutex cannot be locked. The type `TimeT`, used as the argument to `try_lock()`, is defined in the module `TimeBase`.

• **unlock()** resets the mutex state back to unlocked. If there is a single thread waiting to acquire the lock, it will do so at this time. If multiple threads are waiting, and either `SCHED_FIFO` or `SCHED_RR` scheduling policies (described in Table 9-3) are in effect, the mutex is acquired in priority order. If the threads implementation does not support the aforementioned scheduling policies, or a different scheduling policy is used, the order in which threads are awarded the lock is undefined.

RT CORBA Mutexes are supplied by the `RTORB`. The operations which affect the life cycle of Mutexes are shown below:

```plaintext
module RTCORBA
{
  local interface RTORB
  {
    // ...
    Mutex create_mutex ();
    void destroy_mutex (in Mutex the_mutex);
  
    // TAO specific ...
  };
};

• `RTORB::create_mutex()` creates a new instance of a mutex and returns a reference to it. An RT CORBA mutex is a reference-counted object.

• `RTORB::destroy_mutex (in Mutex the_mutex)` cleans up the resources held by the mutex object. In TAO, `destroy_mutex()` removes the mutex from the internal table of named mutexes.

In addition to these RT-CORBA-compliant operations, TAO provides extra functionality. See 9.5.3 for details on the TAO extension.

### 9.3.7 Thread Pools

A **thread pool** is a collection of threads that are all separately available to perform work on behalf of the ORB. A typical thread pool consists of two or
more threads, all waiting for incoming requests. The threads in a pool may be of the same default priority, or they may be grouped together in lanes. Each lane is designated by a certain priority, and each thread in the lane has that priority. Upon creation, a thread pool may have a number of pre-created static threads. A number of dynamic threads may be created later if needed. The RT CORBA specification provides a mechanism that allows a lane of higher priority to borrow a thread from a lane of lesser priority in the same thread pool, if needed.

**Note**  
TAO thread pools are not fully compliant with the RT CORBA specification. Specifically, request buffering and thread borrowing are not supported.

### 9.3.7.1 Interface Specifications

The RTCORBA module defines several types that are used in conjunction with thread pools, as shown here:

```plaintext
module RTCORBA
{
  // Threadpool types.
  typedef unsigned long ThreadpoolId;

  struct ThreadpoolLane
  {
    Priority lane_priority;
    unsigned long static_threads;
    unsigned long dynamic_threads;
  };
  typedef sequence <ThreadpoolLane> ThreadpoolLanes;

  // Threadpool Policy.
  const CORBA::PolicyType THREADPOOL_POLICY_TYPE = 41;
  local interface ThreadpoolPolicy : CORBA::Policy
  {
    readonly attribute ThreadpoolId threadpool;
  };
}
```

Thread pools are identified by a value of type RTCORBA::ThreadpoolId. A POA may be associated with a single thread pool. This is done by supplying a ThreadpoolPolicy as part of the PolicyList supplied to create_POA(). Because a POA may only be associated with a single thread pool, there is only one ThreadpoolId in the ThreadpoolPolicy. The thread pool policy may
also be applied to the ORB to set the default thread pool used by subsequently-
created POAs.

Thread pools are created by the RTORB, using the IDL operations shown here:

```idl
module RTCORBA
{
    local interface RTORB
    {
        // Threadpool creation/destruction.
        exception InvalidThreadpool {};

        ThreadpoolId create_threadpool (in unsigned long stacksize,
                                        in unsigned long static_threads,
                                        in unsigned long dynamic_threads,
                                        in Priority default_priority,
                                        in boolean allow_request_buffering,
                                        in unsigned long max_buffered_requests,
                                        in unsigned long max_request_buffer_size);

        ThreadpoolId create_threadpool_with_lanes (
            in unsigned long stacksize,
            in ThreadpoolLanes lanes,
            in boolean allow_borrowing,
            in boolean allow_request_buffering,
            in unsigned long max_buffered_requests,
            in unsigned long max_request_buffer_size);

        void destroy_threadpool (in ThreadpoolId threadpool)
            raises (InvalidThreadpool);
    }
};
```

### 9.3.7.2 Creating Thread Pools

Thread pools are managed by the ORB. This is similar to using multiple
threads with the thread-pool reactor (see Chapter 20). The main advantage to
using RT CORBA thread pools is that multiple lanes can be created with
differing thread priorities.

Here we show the steps necessary to create a thread pool. First, the thread pool
lanes must be defined. To do this, an RTCORBA::ThreadpoolLanes
sequence is instantiated and filled with information specifying the lane priorities and the number and types of threads to create:

```c++
// Set the thread pool lane size.
CORBA::ULong TOTAL_LANES = // get a value from somewhere
RTCORBA::ThreadpoolLanes lanes(TOTAL_LANES);
lanes.length(TOTAL_LANES);

// Initialize the lane information.
for (CORBA::ULong i = 0; i < TOTAL_LANES; ++i) {
    lanes[i].static_threads = 1;
    lanes[i].dynamic_threads = 0;

    // Initialize the lane_priority (a value between 0 - 32767).
    lanes[i].lane_priority = // some priority value
}
```

Next, use the RTORB to create the thread pool and child POA:

```c++
// Create the threadpool and get back its ThreadpoolId.
RTCORBA::ThreadpoolId threadpool_id =
    rt_orb->create_threadpool_with_lanes(0, // Stack Size
                                          lanes,
                                          0, // Allow borrowing
                                          0, // Allow request buffering
                                          0, // Max buffered requests
                                          0); // Max request buffer size

// Create a policy list.
CORBA::PolicyList poa_policy_list(2);
poa_policy_list.length(2);

// Set the priority model (client propagated for this example).
poa_policy_list[0] =
    rt_orb->create_priority_model_policy(RTCORBA::CLIENT_PROPAGATED, 0);

// Set the thread pool id.
poa_policy_list[1] =
    rt_orb->create_threadpool_policy(threadpool_id);

// Create the child poa with the policy list.
PortableServer::POA_var child_poa = root_poa->create_POA("child_poa",
  poa_manager.in(),
  poa_policy_list);
```
Operations dispatched to servants activated in this new POA will run in one of the threads from the thread pool at a priority requested by the client application. See the example in 9.9 for more details on the use of thread pools.

### 9.3.7.3 Thread Pool Lane Listen Endpoints

Persistent object references require the same endpoint(s) to be used each time the server is run. When thread pools with lanes are required to support persistent object references, you must supply explicit endpoints for each lane. This is accomplished by specifying the `-ORBLaneListenEndpoints` ORB initialization option. This option takes two parameters—the lane identifier and an endpoint specification. The lane identifier is a composite value of the form $n:m$ where $n$ is the thread pool number, starting with 1, and $m$ is the lane index within the thread pool, starting with 0. The endpoint specification parameter is of the form of an ordinary endpoint specification such as may be provided to the `-ORBListenEndpoints` ORB initialization option.

For example, an application creating a thread pool with three lanes might specify the following lane listen endpoints:

```
-ORBLaneListenEndpoints 1:0 iiop://:1234 \
-ORBLaneListenEndpoints 1:1 iiop://:1235 \
-ORBLaneListenEndpoints 1:2 iiop://:1236
```

to define three explicit endpoints for the pool’s lanes. Another ORB initialization option, `-ORBLaneEndpoint`, is an alias for `-ORBLaneListenEndpoints`. See 19.8.14 and 19.8.15 for more information about these options.

### 9.3.8 End-to-End Priority Propagation

One of the biggest challenges in using CORBA for real-time applications is making sure that the priority of an activity is honored by all of the objects and operations involved in carrying out that activity. End-to-end predictability requires that both client and server respect the system-wide priorities during request processing. Furthermore, the system needs to bound the priority inversions and latencies during end-to-end processing. Of course, ultimately the RT ORB relies upon the real-time operating system to schedule threads appropriately. The RT CORBA specification does not attempt to define or dictate real-time OS capabilities.
The RT CORBA specification defines two common priority propagation models: *client-propagated* and *server-declared*. The priority model is selected through the `RTCORBA::PriorityModelPolicy` as follows:

```c
module RTCORBA
{
    // Priority Model Policy.
    const CORBA::PolicyType PRIORITY_MODEL_POLICY_TYPE = 40;

    enum PriorityModel
    {
        CLIENT_PROPAGATED,
        SERVER_DECLARED
    }

    local interface PriorityModelPolicy : CORBA::Policy
    {
        readonly attribute PriorityModel priority_model;
        readonly attribute Priority server_priority;
    }
};
```

The `PriorityModelPolicy` is a *client-exposed* policy, meaning that a client ORB knows what policy is in force and can adjust itself accordingly. As defined in the CORBA messaging specification, the value of this property is communicated through `IOP::ServiceContexts`. In 9.6 and 9.7, we describe how to use the client-propagated and server-declared priority models to specify how priorities are propagated end-to-end across ORB endsystems.

### 9.3.9 Explicit Binding

Frequently, real-time CORBA systems need to explicitly bind object references prior to their first use. To fulfill this requirement, RT CORBA makes use of the operation `CORBA::Object::_validate_connection()` with the appropriate policies set, which preestablishes a connection between the client and server. This operation forces the server to allocate basic resources necessary to service requests on the physical connection used to generate the call.

### 9.3.10 Priority-banded Connections

The server uses the `create_priority_banded_connection_policy()` operation on the RTORB to create priority bands. This operation takes as a parameter a sequence of `PriorityBand` structures called
RTCORBA::PriorityBands. As shown in the IDL below, a PriorityBand structure contains two priorities, low and high. The low priority represents the minimum priority of the band and the high priority represents the maximum priority of the band.

module RTCORBA
{
    // Priorities.
    typedef short NativePriority;
    typedef short Priority;

    // PriorityBandedConnectionPolicy.
    struct PriorityBand
    {
        Priority low;
        Priority high;
    };
    typedef sequence <PriorityBand> PriorityBands;
};

Each band corresponds to one or more lanes within a thread pool on the server. The following example shows how to create priority-banded connection policies.

// Create the sequence of priority bands.
CORBA::ULong NUM_BANDS = // some number of bands
RTCORBA::PriorityBands bands(NUM_BANDS);
bands.length(NUM_BANDS);

// Populate the priority band sequence.
bands[0].low = low_value1;
bands[0].high = high_value1;
bands[1].low = low_value2;
bands[1].high = high_value2;
...

// Create the policy list and add the priority banded connection policy.
CORBA::PolicyList policy_list(1);
policy_list.length(1);
policy_list[0] = rt_orb->create_priority_banded_connection_policy(bands);

// Create a child poa with the priority banded connection policy.
PortableServer::POA_var child_poa = root_poa->create_POA("child_poa",
            poa_manager.in(),
            policy_list);
The priority band chosen depends on the priority model specified by the server. See 9.6 and 9.7 for more information on priority models.

### 9.3.11 Private Connections

Ordinarily, the client ORB is allowed to reuse a connection to support many object references. However, multiplexing requests for different object references on a single connection carries a risk of blocking a thread if the connection is busy during an invocation triggered by another thread. The RT CORBA `PrivateConnectionPolicy` allows the application to specify that dedicated, non-multiplexed connections will be used for certain object references. When this policy is applied to an ORB or a thread, each object reference will have a private connection associated with it. Be aware that the connection is associated with the object reference, not the ORB or thread. If multiple threads use the same object reference, they may still share a connection.

An application should use the `RTORB` to create instances of the `RTCORBA::PrivateConnectionPolicy` policy. This policy may be applied to an `RTORB` via the `ORBPolicyManager`, or to a specific thread via the `PolicyCurrent`. In the following example, we apply the private connection policy to the `PolicyCurrent` to ensure that requests on object references invoked from within each thread will be carried over private connections to the servers hosting the referenced objects.

```c++
CORBA_Object_var obj = orb->resolve_initial_references("PolicyCurrent");
CORBA::PolicyCurrent_var policy_current =
    CORBA::PolicyCurrent::_narrow(obj.in());

// Create a policy list to supply to the PolicyCurrent.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] = rt_orb->create_private_connection_policy();

policy_current->set_policy_overrides (policy_list,
    CORBA::SET_OVERRIDE);
```

### 9.3.12 Protocol Properties

Protocol Properties were introduced into the RT CORBA specification to allow users to specify a preferred protocol to use for connections between clients and servers, and to fine tune the parameters of the physical transport
over which GIOP requests are made. As shown below, the ProtocolProperties interface does not contain any operations or attributes:

```plaintext
module RTCORBA {
    // Protocol Properties.
    local interface ProtocolProperties {
    };
};
```

The TCPProtocolProperties interface corresponds to GIOP over TCP/IP (IIOP) and allows the application to specify the sizes of the TCP send and receive buffers, as well as the TCP keep-alive, routing, and delay attributes. The enable_network_priority attribute is a TAO extension and is described in 9.5.2. The TCPProtocolProperties interface is shown here:

```plaintext
module RTCORBA {
    local interface TCPProtocolProperties : ProtocolProperties {
        attribute long send_buffer_size;
        attribute long recv_buffer_size;
        attribute boolean keep_alive;
        attribute boolean dont_route;
        attribute boolean no_delay;
        attribute boolean enable_network_priority;
    };
};
```

TAO also provides protocol property interfaces for the TAO-specific transport protocols UIOP and SHMIOP as shown below:

```plaintext
module RTCORBA {
    // Communication over Unix Domain Sockets (Local IPC).
    local interface UnixDomainProtocolProperties : ProtocolProperties {
        attribute long send_buffer_size;
        attribute long recv_buffer_size;
    };

    // Communication over Shared Memory.
    local interface SharedMemoryProtocolProperties : ProtocolProperties {
```
RT CORBA specifies two new policies for configuring protocols:

- ServerProtocolPolicy
- ClientProtocolPolicy

When the server or client protocol policy is created, several protocols can be configured at the same time by specifying the protocols and their properties in a sequence called a ProtocolList. The order in which protocols are specified in the list indicates the order of preference. The server-side ORB lists protocol information in this same order in IORs created through that ORB; the client-side ORB considers the protocols in this same order when binding to the server. Server and client protocol policies are created via factory operations on the RT ORB interface. These type and interface definitions are shown below:
The following example shows how to specify protocol properties on the server side. In this example, we configure two protocols—IIOP and the TAO-specific UIOP—with IIOP specified as the preferred protocol.

```cpp
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[])
{
    try
    {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

        // Get a reference to Root POA and activate it.
        CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());
        PortableServer::POAManager_var mgr = poa->the_POAManager();
        mgr->activate();

        // Get the RTORB.
        obj = orb->resolve_initial_references ("RTORB");
        RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());

        // Create a protocol list with 2 elements
        RTCORBA::ProtocolList protocols(2);
        protocols.length(2);

        // Specify the TCP properties.
        CORBA::Long send_buffer_size = 16384;
        CORBA::Long recv_buffer_size = 16384;
        CORBA::Boolean keep_alive = 1;
        CORBA::Boolean dont_route = 0;
        CORBA::Boolean no_delay = 1;

        // Create TCP protocol properties.
        RTCORBA::TCPProtocolProperties_var tcp_properties =
            rtorb->create_tcp_protocol_properties(
                send_buffer_size,
                recv_buffer_size,
            );
```

```cpp
    ServerProtocolPolicy create_server_protocol_policy (in ProtocolList protocols);

    ClientProtocolPolicy create_client_protocol_policy (in ProtocolList protocols);
};
```
Real-Time CORBA

```cpp
keep_alive,
dont_route,
no_delay);

// Specify the TCP (IIOP) protocol as the primary protocol type.
protocols[0].protocol_type = TAG_INTERNET_IOP;
protocols[0].transport_protocol_properties =
   RTCORBA::ProtocolProperties::duplicate (tcp_properties.in ());

// Next, use UIOP with the default values if IIOP fails.
protocols[1].protocol_type = TAO_TAG_UIOP_PROFILE;
protocols[1].transport_protocol_properties =
   RTCORBA::UnixDomainProtocolProperties::nil ();

// Create server protocol policy and insert it into a policy list.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] = rtorb->create_server_protocol_policy (protocols);

// Set the policy on a new child POA.
PortableServer::POA_var child_poa = poa->create_POA ("childPOA", mgr.in (), policy_list);
//...
```

### 9.3.13 Other Real-Time CORBA Features

Other aspects of the RT CORBA specification are covered in other chapters of this guide.

- **Timeouts**
  See 7.3.3 for information on request and reply timeouts.

- **Reliable oneways**
  See 7.3.4 for information on specifying the reliability of `oneway` requests.

- **Asynchronous Invocations**
  Asynchronous invocations of operations on CORBA objects are covered in 7.2.

### 9.4 Dynamic Scheduling

CORBA Real-Time Scheduling is defined in section 3 of the Real-Time CORBA specification, version 1.2 (OMG document formal/05-01-04). This
specification supersedes the static real-time scheduling service defined in version 1.0.

**Note** This specification is also known as the “static” specification. The OMG’s document access page refers to a separate document, formal/03-11-01, as the “dynamic” specification. In fact, version 1.2 supersedes version 2.0, and represents an integration of the dynamic scheduling features first specified in version 2.0 back into the 1.x specification branch.

Static scheduling depends upon knowing the run time characteristics of a system *a priori* in order to determine the scheduling needs. Dynamic scheduling, on the other hand, is much more flexible, allowing for run time selection of scheduling behavior. This is achieved through the implementation of scheduling disciplines. The RT CORBA specification lists some well known disciplines, such as *Fixed Priority Scheduling* (i.e., *Static*) and *Earliest Deadline First*. These disciplines are used to evaluate thread execution order when using a scheduler to dispatch threads. See section 9.4.5 for more information on the scheduler.

A distributable thread is a schedulable entity that maps to an operating system thread while in the context of a particular process, but may pass between processing nodes carrying schedule requirements as service contexts. Distributable threads may be newly spawned, or may be created from an existing thread. Distributable threads carry context information with them as they span nodes. This context information is accessed by the scheduler in each node through the use of portable interceptors.

### 9.4.1 Distributable Threads

The RT CORBA specification defines *distributable threads* as schedulable threads that can run across many nodes in a distributed system. The interface RTScheduling::DistributableThread is used by both the Current and Scheduler interfaces, but may also be used by the application. Distributable thread creation occurs in two ways:

- Creating a new operating system thread by calling
  RTScheduling::Current::spawn(), described in 9.4.3.
• Invoking `begin_scheduling_segment()`, also described in 9.4.3. Calling `begin_scheduling_segment()` when not in a distributable thread makes the current thread distributable.

The distributable thread interfaces shown below provide only the means of canceling the thread while it is running. Distributable threads work in conjunction with *scheduling segments*, which in turn are managed by the `RTScheduling::Current`.

```cpp
module RTScheduling
{
    local interface DistributableThread
    {
        enum DT_State
        {
            ACTIVE,
            CANCELLED
        };

        void cancel();
        readonly attribute DT_State state;
    };
};
```

The `RTScheduling::DistributableThread::cancel()` operation may be used to cancel a running thread. The thread’s current state may be referenced via the `state` attribute. Any thread may cancel a distributable thread, however it is dangerous to try to cancel a thread other than the current thread. While a distributable thread may span many nodes of a distributed system, the interface shown above is local, meaning it only has effect in the current process. It is possible to obtain a reference to a `DistributableThread` object in one process while the actual head of the thread is in another process. Calling `cancel()` on such a `DistributableThread` will then not have the desired effect as the cancellation is indicated as a change of state in the span. While a distributable thread’s `cancel()` operation may be invoked at any time, in any process, doing so does not necessarily alter the thread’s processing.

### 9.4.2 Real-Time Scheduling Thread Action

Thread Action objects, which implement the `RTScheduling::ThreadAction` interface, are the RT CORBA equivalent of thread functions.
module CORBA
{
    native VoidData;
};

module RTScheduling
{
    local interface ThreadAction
    {
        void do (in CORBA::VoidData data);
    };
};

Objects that implement the ThreadAction interface are required to provide a method to be invoked when using RTScheduling::Current::spawn() to start a new distributable thread. The spawn() operation takes a reference to a ThreadAction and invokes do() on it in the native thread creation method. The data argument is a native type, CORBA::VoidData, which is defined as a void* by the C++ language mapping and is used the same way that a void* argument is supplied to a C/C++ thread function.

### 9.4.3 Real-Time Scheduling Current

The real-time scheduling current, RTScheduling::Current, is a specialization of the RTCORBA::Current interface providing additional operations related to creation and management of distributable threads. Through the RTScheduling::Current, distributable threads may be spawned or the current thread may be converted into a distributable thread. A distributable thread may span several scheduling segments. The Current provides the means to identify all the segments of the currently operating thread.

TAO’s definition of RTScheduling::Current deviates slightly from the RT CORBA specification in one important regard. In TAO, the signature of the spawn() operation contains additional parameters that are not present in the RT CORBA specification. The RT CORBA specification defines the spawn() operation as follows:

module RTScheduling
{
    local interface Current : RTCORBA::Current
    {
        // Standard RT CORBA spawn() operation definition.
    };
};
DistributableThread spawn (in ThreadAction start,
in unsigned long stack_size,
in RTCORBA::Priority base_priority);

TAO extends this interface to include parameters that would normally be supplied to the `begin_scheduling_segment()` operation.

module RTScheduling
{
    local interface Current : RTCORBA::Current
    {
        // TAO’s extended spawn() operation definition.
        DistributableThread spawn (in ThreadAction start,
in CORBA::VoidData data,
in string name,
in CORBA::Policy sched_param,
in CORBA::Policy implicit_sched_param,
in unsigned long stack_size,
in RTCORBA::Priority base_priority);
    }
};

TAO’s extended `spawn()` operation combines two steps into one. The RT CORBA specification is very open in terms of how distributable threads may interact with schedules. It states that schedules, or schedulers, are not required, but does not completely describe what should happen when they are not present. By allowing distributable threads to exist outside of schedules, the specification imposes an explicit burden for any thread to also invoke `begin_scheduling_segment()` when starting. This requires that the schedule segment information either be known implicitly by the thread function, or it must be communicated to the thread function via the `data` parameter, which usually carries application data for the thread. TAO’s implementation of `spawn()` carries the schedule segment information separately from any application-specific thread data. It also uses a wrapper function to invoke `begin_scheduling_segment()` before, and `end_scheduling_segment()` after, invoking the thread function.

The `RTScheduling::Current` is responsible for managing scheduling segments. Threads invoke `begin_scheduling_segment()` to start a new, and possibly nested, scheduling segment. Threads invoke `end_scheduling_segment()` at the completion of a scheduling segment.
9.4 Dynamic Scheduling

Threads may also invoke `update_scheduling_segment()` to modify the attributes of the current scheduling segment, if necessary.

```idl
module RTScheduling
{
    local interface Current : RTCORBA::Current
    {
        exception UNSUPPORTED_SCHEDULING_DISCIPLINE {};

        void begin_scheduling_segment (in string name,
            in CORBA::Policy sched_param,
            in CORBA::Policy implicit_sched_param)
            raises (UNSUPPORTED_SCHEDULING_DISCIPLINE);

        void update_scheduling_segment (in string name,
            in CORBA::Policy sched_param,
            in CORBA::Policy implicit_sched_param)
            raises (UNSUPPORTED_SCHEDULING_DISCIPLINE);

        void end_scheduling_segment (in string name);
    }
};
```

The operations above allow RT CORBA applications to manage scheduling segments. Scheduling segments may be named, and the name can be used when ending a scheduling segment to ensure the correct segment is ended; otherwise, the name has no purpose. The `sched_param` and `implicit_sched_param` parameters are used to describe how a scheduling segment relates to other scheduling segments within the scheduler. The `sched_param` parameter is used to explicitly define the new or updated scheduling segment. The `implicit_sched_param` parameter is used whenever a nested scheduling segment is started with a nil `sched_param` value. There are no constraints placed on the definition of the scheduling parameters by the RT CORBA specification other than the fact that they must derive from the `CORBA::Policy` interface. Scheduling parameters are created by schedulers that implement a particular scheduling discipline, by way of a factory operation that returns scheduling parameters specific to that discipline. The RT CORBA specification describes a number of well-known scheduling disciplines, and includes example IDL specifications for each. These scheduling disciplines are briefly described in section 9.4.5. See also section 3.7 of the RT CORBA 1.2 specification for a more complete description of these scheduling disciplines.
9.4.4 Real-Time Scheduling Resource Manager

The RT CORBA specification defines a scheduler-aware specialization of the RTCORBA::Mutex called RTScheduling::ResourceManager. This interface defines no new operations or attributes, it simply provides a means to distinguish a resource manager from a base mutex. A ResourceManager is created by calling the create_resource_manager() operation on the RTScheduling::Scheduler interface. See 9.4.5 for more details on the RTScheduling::Scheduler interface. Operations that acquire (RTScheduling::ResourceManager::lock() or RTScheduling::ResourceManager::try_lock()) or release (RTScheduling::ResourceManager::unlock()) a real-time CORBA resource manager are defined as scheduling points, meaning the scheduler will have a chance to run and reassess the scheduling parameters, thereby ensuring that the scheduling discipline is maintained. Distributable threads may share local resources by using RTScheduling::ResourceManager operations.

9.4.5 Real-Time Scheduling Scheduler

A scheduler is responsible for allocating system resources and determining timeliness. The real-time CORBA scheduler is intended to be pluggable. ORB implementations may provide different scheduler implementations as long as they implement the RTScheduling::Scheduler interface, defined in RTScheduler.pidl. This pluggable architecture affords implementors the flexibility to provide specific schedulers, each with particular scheduling characteristics. The scheduler works in conjunction with portable interceptors to track scheduling segments of a distributable thread. The real-time scheduler interface is described in 9.4.5.3, where it is presented in context with the other elements of real-time scheduling.

9.4.5.1 TAO's RTScheduleManager

While the real-time CORBA specification describes the RTScheduling::Scheduler interface and its role in distributed real-time systems, it does not mandate an implementation or even a scheme for managing schedulers. The specification states that using a scheduler is optional, and for ORBs that have a scheduler, it may be accessed by calling resolve_initial_references("RTScheduler"). TAO, on the other hand, provides a TAO-specific initial reference called the "RTScheduleManager". A call to
resolve_initial_references("RTScheduleManager") returns an object through which an application may supply its own scheduler.

9.4.5.2 Scheduling Disciplines
A scheduler is used to implement a particular scheduling discipline. It may implement a well-known scheduling discipline such as Earliest Deadline First or something esoteric such as Most Important First. The real-time CORBA specification describes four well-known scheduling disciplines:

- Fixed Priority
- Earliest Deadline First (EDF)
- Least Laxity First (LLF)
- Maximize Accrued Utility (MAU)

For each of these disciplines, the specification provides a recommended interface, including the definition of the particular scheduling parameter type and how it might be created.

In addition, TAO provides an example scheduler that implements a scheduling discipline that is not one of the above well-known disciplines, but that may be useful in some situations:

- Most Important First (MIF)

The following sections briefly describe each of the well-known scheduling disciplines, the associated scheduling parameter types, and the scheduler interface.

9.4.5.3 Common RTScheduling::Scheduler Interface
Each of the discipline-specific schedulers listed above extends the common base interface RTScheduling::Scheduler. The RT CORBA specification provides an interface for this base type. TAO extends the Scheduler interface, as shown in 9.5.4. The RT CORBA specified interface focuses on resource management and is shown here:

```cpp
module RTScheduling
{
  local interface Scheduler
  {
    exception INCOMPATIBLE_SCHEDULING_DISCIPLINES {};
  }
}
```
attribute CORBA::PolicyList scheduling_policies;
readonly attribute CORBA::PolicyList poa_policies;
readonly attribute string scheduling_discipline_name;

ResourceManager create_resource_manager (  
    in string name,
    in CORBA::Policy scheduling_parameter);

void set_scheduling_parameter (  
    inout PortableServer::Servant resource,
    in string name,
    in CORBA::Policy scheduling_parameter);

Note that the exception INCOMPATIBLE_SCHEDULING_DISCIPLINES is not
explicitly referenced by any of the operations defined in the specification.
The scheduling_policies and poa_policies attributes provide the
scheduler with a way of listing any POA policies that might be required for the
scheduler to work. It is reasonable and typical for these attributes to return nil
values.
The scheduling_discipline_name attribute is simply a name that may be
queried by the application to identify the particular discipline implemented by
the scheduler.
The create_resource_manager() operation is used to create instances of
RTScheduling::ResourceManager, providing it with a name and
optionally associating it with a scheduling parameter.
The set_scheduling_parameter() operation. This operation is intended
to be a hook giving schedulers a way to associate certain schedule parameters,
such as a fixed priority ceiling, to a particular resource. The resource in this
case is a servant.

It is not necessary for a scheduler to implement all or any of these operations.
They are defined merely as hooks to provide scheduler implementors a means
to express any sort of scheduling needs in code.

### 9.4.5.4 Fixed Priority Scheduling
The fixed priority scheduling discipline, also known as rate-monotonic
scheduling, is characterized by static schedules where the schedule is
completely determined prior to deployment. An example of fixed priority
Dynamic Scheduling

The fixed priority scheduler is a suitable replacement for the older static scheduling model.

Fixed priority scheduling is defined in the FP_Scheduling module. The fixed priority scheduling parameter is defined by the FP_Scheduling::SegmentSchedulingParameterPolicy local interface. It has an attribute representing a single RT CORBA priority value. The fixed priority scheduler is defined by the FP_Scheduling::FP_Scheduler interface.

```cpp
module FP_Scheduling
{
    local interface SegmentSchedulingParameterPolicy : CORBA::Policy
    {
        attribute RTCORBA::Priority value;
    };

    local interface FP_Scheduler : RTScheduling::Scheduler
    {
        SegmentSchedulingParameterPolicy create_segment_scheduling_parameter (in RTCORBA::Priority segment_priority);
    };
};
```

**9.4.5.5 Earliest Deadline First Scheduling**

The *earliest deadline first* (EDF) scheduling discipline places emphasis on those threads that must complete the soonest. EDF scheduling segments are based on a required completion time, which may be modified further by an indication of a thread’s importance depending upon a particular scheduler’s implementation. TAO does not currently provide a reference implementation of the EDF scheduling discipline. The RT CORBA specification, however, provides a suggested EDF scheduling interface in the EDF_Scheduling module, shown here:

```cpp
module EDF_Scheduling
{
    struct SchedulingParameter
    {
        TimeBase::TimeT deadline;
        long importance;
    };
};
```
local interface SchedulingParameterPolicy : CORBA::Policy {
    attribute SchedulingParameter value;
};

local interface Scheduler : RTScheduling::Scheduler {
    SchedulingParameterPolicy create_scheduling_parameter ( 
        in SchedulingParameter value);
};

Note that the deadline member of the scheduling parameter is of type TimeBase::TimeT. This type is defined as part of the CORBA Time Service specification.

9.4.5.6 Least Laxity First Scheduling

The least laxity first (LLF) scheduling discipline favors threads that have the least amount of time they can wait before missing a deadline (i.e., being late). The real-time CORBA specification defines laxity as:

\[
\text{laxity} = \text{deadline} - \text{current time} - \text{estimated remaining time to completion}
\]

In the LLF scheduling discipline, threads that have a lower laxity value are scheduled first.

Least laxity first scheduling is defined in the LLF_Scheduling module:

module LLF_Scheduling {
    struct SchedulingParameter {
        TimeBase::TimeT deadline;
        TimeBase::TimeT estimated_initial_execution_time;
        long importance;
    };

    local interface SchedulingParameterPolicy : CORBA::Policy {
        attribute SchedulingParameter value;
    };

    local interface Scheduler : RTScheduling::Scheduler {
        SchedulingParameterPolicy create_scheduling_parameter ( 

The LLF scheduling parameter is similar to the EDF scheduling parameter in that it includes a deadline and an importance qualifier. An estimate of the amount of time this schedule segment is anticipated to take to complete is also included, so that the scheduler can compute the laxity for this segment. TAO does not currently provide a reference implementation of the LLF scheduling discipline.

9.4.5.7 Maximize Accrued Utility Scheduling

The maximize accrued utility (MAU) scheduling discipline is a special case of the Earliest Deadline First scheduling discipline. In MAU scheduling, a special function, called the utility function, is used to compute a utility value for a thread. The utility of a thread is a measure of the likelihood that a thread will complete close to, but prior than, its deadline. A thread that can complete close to its deadline has a greater utility value than a thread completing much earlier. A thread that will complete after its deadline may have a zero or negative utility value. The MAU discipline seeks a schedule that results in maximal accrued (i.e., summed) utility.

Maximize accrued utility scheduling is defined in the Max_Utility_Scheduling module:
The MAU scheduling parameter is similar to the EDF scheduling parameter in that it includes a deadline and an importance qualifier. TAO does not currently provide a reference implementation of the MAU scheduling discipline.

9.4.5.8 **Most Important First Scheduling**

TAO provides an example of a scheduler that implements a somewhat different scheduling discipline from the well-known disciplines described above. The *most important first* (MIF) scheduler makes use of a thread’s *importance* to determine which thread should execute. The MIF scheduler interface is defined in the MIF_Scheduling module in $TAO_ROOT/examples/RTScheduling/MIF_Scheduling.pidl.

```plaintext
module MIF_Scheduling
{
    local interface SegmentSchedulingParameterPolicy : CORBA::Policy
    {
        attribute short importance;
    };

    local interface MIF_Scheduler : RTScheduling::Scheduler
    {
        SegmentSchedulingParameterPolicy create_segment_scheduling_parameter (in short segment_importance);
    };
};
```

An implementation of MIF scheduling is provided in $TAO_ROOT/examples/RTScheduling/MIF_Scheduler.

9.5 **TAO’s Implementation of Real-Time CORBA**

This section provides details on using TAO’s implementation of RT CORBA and RT CORBA extensions provided in TAO. Topics in this section include:

- Priority Mapping in TAO
- Enabling Network Priority in TAO
- Using TAO’s Named Mutexes
- Building RT CORBA Support into TAO
• Configuring the RT ORB Component

9.5.1 Priority Mapping In TAO
TAO offers three priority mappings: continuous, direct and linear. These priority mappings are explained in the following subsections.

9.5.1.1 Continuous Priority Mapping
The continuous priority mapping, as shown in Figure 9-3, uses only the first $n$ priorities of CORBA’s priority range, providing a one-to-one mapping of native-to-CORBA priorities, where $n$ is the number of discrete native priority

![Figure 9-3 TAO Continuous Mapping Behavior](image-url)
values permitted by the operating system. The lowest native priority maps to a CORBA priority value of 0, the next higher native priority maps to 1, etc.

Because the mapping functions, `PortableMapping::to_native()` and `PortableMapping::to_CORBA()` are idempotent, you can start with a priority value, convert it twice and end up with the same value. This advantage of continuous mapping is countered by the disadvantage that part of the CORBA priority range is invalid. If you are using more than one RTOS in a distributed environment, where each RTOS defines a different number of discrete priority values, some CORBA priorities will map to valid native priorities in one RTOS, but be invalid in another RTOS.

### 9.5.1.2 Direct Priority Mapping

The direct priority mapping is similar to the continuous priority mapping in that some of the CORBA priority values are invalid. However, native priority values are passed directly through as CORBA priority values. As shown in Figure 9-4, no priority transform is applied to the native value. This mapping would only be desirable where native priority values fall within the minimum and maximum values of the CORBA priority range.

It is possible to have native priority values that do not directly map onto the CORBA priority range. The direct mapping would be useless in this situation. An attempt to set the priority from a native value that does not map into
CORBA priority range would cause a `CORBA::DATA_CONVERSION` exception to be raised.

9.5.1.3 Linear Priority Mapping

The linear priority mapping is one-to-many, providing a range of CORBA priority values for each native priority value. The size of the range is 

\[
\frac{(RTCORBA::maxPriority - RTCORBA::minPriority)}{n}
\]

where \( n \) is the number of discrete native priorities. For instance, a native system offering 16
priority values would result in a map with 2048 CORBA priority values for each native priority value.

The risk with the linear priority mapping is that rounding can occur. In other words, a CORBA priority converted to a native priority may be converted back to a different CORBA priority value than the original. Furthermore, if objects are hosted on processes using different real-time operating systems with sufficiently different priority ranges, the linear mapping may result in a native priority being communicated, then communicated back as a different native priority.

**Figure 9-5 TAO Linear Mapping Behavior**

The risk with the linear priority mapping is that rounding can occur. In other words, a CORBA priority converted to a native priority may be converted back to a different CORBA priority value than the original. Furthermore, if objects are hosted on processes using different real-time operating systems with sufficiently different priority ranges, the linear mapping may result in a native priority being communicated, then communicated back as a different native priority.

### 9.5.1.4 Using TAO’s Priority Mappings

The RT CORBA specification only defines what a priority-mapping object is to do, not how it is to do it. Furthermore, the specification does not define a
means of accessing the priority-mapping object. What is defined is RTScheduling, which hides the details of priority mapping from the application.

To ensure that priority mappings are used consistently in an environment, TAO is configured at run time through the \texttt{RT\_ORB\_Loader} service configuration option \texttt{ORBPriorityMapping} (see 9.5.7). Within your application, the ORB’s priority-mapping object may be obtained through a helper object, an instance of \texttt{TAO\_Priority\_Mapping\_Manager} (or its alias \texttt{RTCORBA::PriorityMappingManager}) as shown below:

```cpp
class TAO_RTCORBA_Export TAO_Priority_Mapping_Manager :
    public virtual TAO_Local_RefCounted_Object
{
public:
//... implementation details not shown

    // Get the current priority mapping.
    RTCORBA::PriorityMapping* mapping (void);

    // Set a new priority mapping.
    void mapping (RTCORBA::PriorityMapping* new_mapping);
};
```

Since the priority-mapping manager, \texttt{TAO\_Priority\_Mapping\_Manager}, inherits from \texttt{TAO\_Local\_RefCounted\_Object}, it ultimately inherits from \texttt{CORBA::Object}, which means that a reference to it may be obtained from the ORB via a call to \texttt{resolve\_initial\_references()}. The name used for this object is \texttt{PriorityMappingManager}. The object reference returned from \texttt{resolve\_initial\_references()} must be narrowed using a call to \texttt{RTCORBA::PriorityMappingManager::\_narrow()}. The \texttt{mapping()} operation is then called on the resulting object reference to obtain a pointer to the \texttt{RTCORBA::PriorityMapping} object as shown below:

```cpp
CORBA::Object_var obj =
    orb->resolve_initial_references ("PriorityMappingManager");
RTCORBA::PriorityMappingManager_var mapping_manager =
    RTCORBA::PriorityMappingManager::\_narrow (obj.in());
RTCORBA::PriorityMapping* priority_mapping = mapping_manager->mapping();
```

Note that the \texttt{PriorityMappingManager} retains ownership of the \texttt{PriorityMapping}, so it should not be deleted.
9.5.2 Enabling Network Priority in TAO

Some environments provide support for differentiated classes of network service, and allow applications to specify their network quality of service needs. A common mechanism for providing differentiated classes of service on IP networks is the Differentiated Services (diffserv) architecture defined by the Internet Engineering Task Force (IETF) Diffserv Working Group. In the diffserv architecture, applications encode a particular six-bit pattern into a field, called the DS field, of the IP packet header, thereby marking a packet to receive a particular forwarding treatment, or per-hop behavior (PHB), at each network node. The Diffserv Working Group has standardized a small number of specific per-hop behaviors and a recommended bit pattern, or codepoint, for each one. These PHBs and their recommended codepoints are defined in various IETF Requests for Comments (RFCs). For more information on Differentiated Services and Diffserv Codepoints (DSCPs), see RFC 2474, RFC 2475, RFC 2597, RFC 2598, and RFC 3246, all of which are available from the IETF at <http://www.ietf.org/rfc/>.

The RT CORBA specification does not provide a way to map RT CORBA priorities to network priorities. As an extension to RT CORBA, TAO provides a mechanism to map RT CORBA priorities to network priorities via diffserv codepoints. Applications enable this mapping via a TAO-specific extension to the TCPProtocolProperties described in 9.3.12. The interface is:

```plaintext
module RTCORBA
{
    local interface TCPProtocolProperties : ProtocolProperties
    {
        attribute long send_buffer_size;
        attribute long recy_buffer_size;
        attribute boolean keep_alive;
        attribute boolean dont_route;
        attribute boolean no_delay;
        attribute boolean enable_network_priority;
    };
};
```

When the enable_network_priority attribute is set to TRUE, mapping between RT CORBA priority and a corresponding network priority diffserv codepoint is enabled. The diffserv codepoint resulting from this mapping is encoded into the DS field in the IP packet header for GIOP requests and replies.
9.5.2.1 Enabling Network Priority in a Client

If you enable network priorities on the client side, the RT CORBA priority of the invoking thread is mapped to a corresponding diffserv codepoint and set in the IP packet header for GIOP requests. To enable network priorities on the client side, you must create a TCPProtocolProperties object, set the enable_network_priority attribute to TRUE, create a protocol properties policy, then set the protocol properties policy at the ORB, thread, or object level. An example of how to do this is shown below:

```c++
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[]) 
{
    try 
    { 
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

        // Get the RTORB.
        CORBA::Long send_buffer_size = 16384;
        CORBA::Long recv_buffer_size = 16384;
        CORBA::Boolean keep_alive = 1;
        CORBA::Boolean dont_route = 0;
        CORBA::Boolean no_delay = 1;

        // Create TCP protocol properties.
        RTCORBA::TCPProtocolProperties_var tcp_properties =
            rtcorb->create_tcp_protocol_properties(
                send_buffer_size,
                recv_buffer_size,
                keep_alive,
                dont_route,
                no_delay);

        // Enable network priority.
        tcp_properties->enable_network_priority (1);

        // Create a protocol list and set the ProtocolProperties.
        RTCORBA::ProtocolList protocols;
        protocols.length (1);
        protocols[0].protocol_type = TAO_TAG_IIOP_PROFILE;
        protocols[0].transport_protocol_properties =
            RTCORBA::ProtocolProperties::_duplicate (tcp_properties.in());
    }
}
```
// Create client protocol policy and insert it into a policy list.
CORBA::PolicyList policy_list;
policy_list.length (1);
policy_list[0] = rtorb->create_client_protocol_policy (protocols);

// Set the policy at the ORB level.
CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
    CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides(policy_list, CORBA::SET_OVERRIDE);

//...

In the example above, since the client protocol policy is set at the ORB level (by setting the policy overrides on the ORBPolicyManager), network priority mapping will be enabled for all requests invoked through that ORB.

### 9.5.2.2 Enabling Network Priority in a Server

If you enable network priorities on the server side, the RT CORBA priority of the request-dispatching thread is mapped to a corresponding diffserv codepoint and set in the IP packet header for the GIOP reply. To enable network priorities on the server side, create TCPProtocolProperties and set the enable_network_priority attribute to TRUE. Then, create a protocol-properties policy and set the policy on a new POA upon creation. An example of how to do this is shown below:

```c++
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[])
{
    try
    {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

      // Get a reference to Root POA and activate it.
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());
      PortableServer::POAManager_var mgr = poa->the_POAManager();
      mgr->activate();

      // Get the RTORB.
      obj = orb->resolve_initial_references("RTORB");
      RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());
```
// Specify the TCP properties.
CORBA::Long send_buffer_size = 16384;
CORBA::Long recv_buffer_size = 16384;
CORBA::Boolean keep_alive = 1;
CORBA::Boolean dont_route = 0;
CORBA::Boolean no_delay = 1;

// Create TCP protocol properties.
RTCORBA::TCPProtocolProperties_var tcp_properties =
  rtorb->create_tcp_protocol_properties(
    send_buffer_size,
    recv_buffer_size,
    keep_alive,
    dont_route,
    no_delay );

// Enable network priority.
tcp_properties->enable_network_priority (1);

// Create a protocol list and set the ProtocolProperties.
RTCORBA::ProtocolList protocols;
protocols.length (1);
protocols[0].protocol_type = TAO_TAG_IIOP_PROFILE;
protocols[0].transport_protocol_properties =
  RTCORBA::ProtocolProperties::_duplicate (tcp_properties.in ());

// Create server protocol policy and insert it into a policy list.
CORBA::PolicyList policy_list;
policy_list.length (1);
policy_list[0] = rtorb->create_server_protocol_policy (protocols);

// Set the policy on a new child POA.
PortableServer::POA_var child_poa = poa->create_POA ( 
  "childPOA", PortableServer::POAManager::_nil(), policy_list);

// ...

In the example above, network priority mapping will be enabled in all replies generated from servants activated in the child POA. Replies generated from servants activated in the Root POA will not have network priority mapping enabled, so take care when using _this to get object references from servants.

9.5.2.3 Network Priority Mappings

Different mappings of RT CORBA priorities to network priorities are possible. A specific network priority mapping is provided via a TAO-specific
NetworkPriorityMapping interface. This mapping is similar to the RT CORBA priority to native priority mapping described in 9.3.4. The interface is shown below:

```cpp
module RTCORBA
{
    typedef long NetworkPriority;
    native NetworkPriorityMapping;
};
```

The C++ binding for NetworkPriorityMapping is a class in the RTCORBA namespace as shown below:

```cpp
namespace RTCORBA {
    class NetworkPriorityMapping {
    public:
        virtual CORBA::Boolean to_network (RTCORBA::Priority corba_priority,
                                               RTCORBA::NetworkPriority& network_priority );
        virtual CORBA::Boolean to_CORBA (RTCORBA::NetworkPriority network_priority,
                                          RTCORBA::Priority& corba_priority );
    };
};
```

The functions `to_network()` and `to_CORBA()` may be called several times by an ORB during an invocation. To provide the greatest possible efficiency these functions do not throw exceptions, not even CORBA System exceptions. However these functions will return `FALSE` if the input value is outside the allowed range for that type. For both of these functions, the first argument has the semantics of an `in`, supplying input, with the second argument being an `out` for receiving the converted results.

During an invocation, the RT ORB uses these mapping functions to map between RT CORBA priority and network priority. If the call to either mapping function returns `FALSE`, the ORB stops processing the invocation and throws a `DATA_CONVERSION` system exception to the invoking application.

By default, TAO uses a linear network-priority mapping that maps RT CORBA priority values to discrete diffserv codepoints recommended by various IETF RFCs. For example, RFC 2474 recommends specific codepoints for “default” per-hop behaviors and various Class Selector (CS) codepoints; RFC 2597 recommends specific codepoints for various Assured Forwarding
(AF) PHBs; RFC 3246 recommends a specific codepoint for Expedited Forwarding (EF) PHB. See the relevant RFCs for more information on the values and semantics of these PHBs and their recommended diffserv codepoints.

### 9.5.2.4 Using TAO’s Network Priority Mappings

Similar to the run-time configuration of priority mappings described in 9.5.1.4, the network priority mapping used by TAO can be configured at run time through the `RT_ORB_Loader` option `ORBNetworkPriorityMapping`, described in 9.5.7. Within your application, the ORB’s network priority-mapping object may be obtained through either a helper object, an instance of `TAO_Network_Priority_Mapping_Manager` or its typedef `RTCORBA::NetworkPriorityMappingManager`.

```cpp
class TAO_RTCORBA_Export TAO_Network_Priority_Mapping_Manager : 
    public virtual TAO_Local_RefCounted_Object 
{
    public:
    //... implementation details not shown

    /// Get the current network priority mapping.
    RTCORBA::NetworkPriorityMapping* mapping (void);

    /// Set a new network priority mapping.
    void mapping (RTCORBA::NetworkPriorityMapping* new_mapping);

};
```

*Because TAO_Network_Priority_Mapping_Manager is derived from TAO_Local_RefCounted_Object, it ultimately is derived from CORBA::Object. As a descendant of CORBA::Object, a reference to the network priority mapping manager may be obtained from the ORB via a call to `resolve_initial_references()`*. The name used for this object is `NetworkPriorityMappingManager`. The object reference returned from `resolve_initial_references()` must be narrowed using a call to `RTCORBA::NetworkPriorityMappingManager::_narrow()`. Then, a call must be made to the `mapping()` function on the resulting object reference to obtain a pointer to the `RTCORBA::NetworkPriorityMappingManager` object. An example of how to do this is shown below:

```cpp
CORBA::Object_var obj = 
    orb->resolve_initial_references ("NetworkPriorityMappingManager");
```
RTCORBA::NetworkPriorityMappingManager_var network_mapping_manager =
    RTCORBA::NetworkPriorityMappingManager::_narrow (obj.in());
RTCORBA::NetworkPriorityMapping* network_priority_mapping =
    network_mapping_manager->mapping();

Note that the NetworkPriorityMappingManager retains ownership of the
NetworkPriorityMapping, so it should not be deleted.

9.5.2.5 Implementing a Custom Network Priority Mapping

An application can implement a custom NetworkPriorityMapping by
deriving a new class from RTCORBA::NetworkPriorityMapping and
overriding the to_network() and to_CORBA() functions as shown below:

class CustomNetworkPriorityMapping :
    public virtual RTCORBA::NetworkPriorityMapping
{
    public:
        virtual CORBA::Boolean to_network (RTCORBA::Priority corba_priority,
            RTCORBA::NetworkPriority& network_priority );
        virtual CORBA::Boolean to_CORBA (RTCORBA::NetworkPriority network_priority,
            RTCORBA::Priority& corba_priority );
    }

CORBA::Boolean CustomNetworkPriorityMapping::to_network (  
    RTCORBA::Priority corba_priority,
    RTCORBA::NetworkPriority& network_priority )
{
    network_priority = // map corba_priority to network_priority
    return 1;
}

CORBA::Boolean CustomNetworkPriorityMapping::to_CORBA (  
    RTCORBA::NetworkPriority network_priority,
    RTCORBA::Priority& corba_priority )
{
    corba_priority = // map network_priority to corba_priority
    return 1;
}

To create an instance of our custom network priority mapping and instruct
TAO to use it instead of the default network priority mapping, we do the
following:

CustomNetworkPriorityMapping* new_network_priority_mapping =
    new CustomNetworkPriorityMapping();
network_mapping_manager->mapping(new_network_priority_mapping);
Note that the NetworkPriorityMappingManager takes ownership of the new network priority-mapping object.

### 9.5.3 Using TAO’s Named Mutexes

The basic RT CORBA Mutex specification only requires that an RTORB create and destroy mutexes. TAO adds the option of maintaining a table of mutexes keyed by a name, freeing the application developer from managing references to the mutex as shown below:

```c
module RTCORBA
{
    local interface RTORB
    {
        // ...
        // TAO specific
        // Named Mutex creation/opening
        exception MutexNotFound {};
        Mutex create_named_mutex (in string name,
                                  out boolean created_flag);
        Mutex open_named_mutex (in string name)
            raises (MutexNotFound);
    }
};
```

In addition to supplying the RT-CORBA-compliant implementation of mutexes, TAO provides the following functions for accessing mutexes by name:

- `RTORB::create_named_mutex(in string name, out boolean created_flag)`

  Creates a new instance of a mutex and returns a reference to it, or returns a reference to an existing mutex of that name. The value of `created_flag` will be TRUE only if the mutex was created as a result of this call.

- `RTORB::open_named_mutex(in string name)`

  Returns a reference to an existing mutex only if it exists. Raises the TAO-specific exception `MutexNotFound` if the supplied name does not match any mutex.
9.5.4 Dynamic Scheduling and TAO

TAO 1.4a fully supports the scheduling feature of RT CORBA. TAO deviates from the specification in a few important respects. In particular, the definitions of the `RTScheduling::Current` (see 9.4.3) and the base interface for the `RTScheduling::Scheduler` (see 9.4.5) are slightly different in TAO than in the specification.

TAO does not provide an implementation of `RTScheduling::Scheduler`. As discussed in 9.4.5, the scheduler is effectively an abstract interface, the real work of scheduling being handled by specialized schedulers. TAO does provide the framework for building your own scheduler and using it to manage the dispatching of distributable threads. TAO’s framework includes a set of portable interceptors to provide message notification to the scheduler. These interceptors provide additional scheduling points in addition to the usual begin, update, and end of a scheduling segment.

TAO extends the base scheduler as defined in the RT CORBA specification to define several operations used by the portable interceptors or `RTScheduling::Current` as schedule evaluation points. TAO’s scheduler interface definition is shown here:

```plaintext
module RTScheduling
{
  local interface Scheduler
  {
    void begin_new_scheduling_segment (in Current::IdType guid,
                                          in string name,
                                          in CORBA::Policy sched_param,
                                          in CORBA::Policy implicit_sched_param)
      raises (Current::UNSUPPORTED_SCHEDULING_DISCIPLINE);
    void begin_nested_scheduling_segment (in Current::IdType guid,
                                           in string name,
                                           in CORBA::Policy sched_param,
                                           in CORBA::Policy implicit_sched_param)
      raises (Current::UNSUPPORTED_SCHEDULING_DISCIPLINE);
    void update_scheduling_segment (in Current::IdType guid,
                                    in string name,
                                    in CORBA::Policy sched_param,
                                    in CORBA::Policy implicit_sched_param)
      raises (Current::UNSUPPORTED_SCHEDULING_DISCIPLINE);
    void end_scheduling_segment (in Current::IdType guid,
                                 in string name,
                                 in CORBA::Policy sched_param,
                                 in CORBA::Policy implicit_sched_param)
      raises (Current::UNSUPPORTED_SCHEDULING_DISCIPLINE);
  }
}
```
9.5 TAO’s Implementation of Real-Time CORBA

```c++

in string name);

void end_nested_scheduling_segment (in Current::IdType guid,
in string name,
in CORBA::Policy outer_sched_param);

}

The scheduler’s operations are invoked by the RTScheduling::Current
whenever a new base or nested schedule segment is begun or ended, or when a
segment is updated. These invocations provide the scheduler an opportunity to
raise an exception (e.g., if an inappropriate exception is used) or to possibly
dispatch or otherwise order existing distributable threads.

The TAO scheduler provides implementations for all of the portable
interceptor interception points. With the exception of receive_request(),
the signatures of all the interception operations is the same as that of the
equivalent PortableInterceptor::ServerInterceptor or
PortableInterceptor::ClientInterceptor operations. The difference
in the signature of the receive_request() operation results from the need
to supply additional information to the scheduler, as shown here:

module RTScheduling
{
  local interface Scheduler
  {
    void receive_request (in PortableInterceptor::ServerRequestInfo ri,
in Current::IdType guid,
in string name,
in CORBA::Policy sched_param,
in CORBA::Policy implicit_sched_param)
      raises (PortableInterceptor::ForwardRequest);
  }
}

As with the common scheduler operations shown in 9.4.5.3, TAO’s additional
operations provide additional hooks to allow the expression of any sort of
scheduler that may be required by the application. The TAO-specific
scheduler interface is intended to describe all possible schedule evaluation
points, making up for apparent deficiencies in the RT CORBA specification’s
scheduler interface.
9.5.5 Enabling RT CORBA Support in TAO
RT CORBA features are controlled when building TAO by the `rt_corba`, `corba_messaging`, and `minimum_corba` build flags. The RT CORBA features are enabled in TAO by default because `rt_corba` is set to 1, `corba_messaging` is set to 1, and `minimum_corba` is set to 0 by default in the TAO make files. If any of these three flags is set to the opposite value, RT CORBA support will not be available in the TAO libraries. The preprocessor macros `TAO_HAS_RT_CORBA`, `TAO_HAS_CORBA_MESSAGING`, and `TAO_HAS_MINIMUM_CORBA` can also be used to set these flags (for example, in build environments where you are not using GNU Make). See A.3 for more information concerning these flags and macros.

9.5.6 Building Applications that use RT CORBA
TAO’s support of basic RT CORBA features is implemented in the `TAO_RTCORBA` library, and the `RTPortableServer` module’s features are implemented in the `TAO_RTPortableServer` library. Thus, applications that use TAO’s RT CORBA features must link with one or both of these libraries. MPC projects for clients that use RT CORBA can simply inherit from the `rt_client` base project. MPC projects for servers that use RT CORBA can simply inherit from the `rt_server` base project. For example, here is the MPC file for the RT CORBA example in `$TAO_ROOT/DevGuideExamples/RTCORBA`:

```cpp
project(*Server): rt_server {
    requires += exceptions
    Source_Files {
        Messenger_i.cpp
        MessengerServer.cpp
        common.cpp
    }
}
```

```cpp
project(*Client): rt_client {
    requires += exceptions
    Source_Files {
        MessengerC.cpp
        MessengerClient.cpp
        common.cpp
    }
}
```
9.5.7 Configuring RT CORBA at Run Time

Certain behavioral aspects of TAO’s implementation of RT CORBA can be configured at run time.

9.5.7.1 RT ORB Loader

The RT ORB can be configured via the `RT_ORB_Loader` service object. The TAO `RT_ORB_Loader` takes initialization options that control the priority mapping type, the network priority mapping type, the scheduling policy, and the thread scoping policy.

The `RT_ORB_Loader` object is initialized by supplying a service configuration directive, typically as a line in a `svc.conf` file. Service configuration directives are explained in further detail in Chapter 18. For example, an application that statically links the `RTCORBA` library may use a static directive as shown here. (The entire directive should appear on one line in the file.)

```
static RT_ORB_Loader "-ORBPriorityMapping linear -ORBSchedPolicy SCHED_FIFO -ORBScopePolicy SYSTEM"
```

The possible configuration options for the `RT_ORB_Loader` service object are listed in the following tables.

**Table 9-1 Priority Mapping Selection Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ORBPriorityMapping mapping_type</code></td>
<td>Selects the algorithm to use when mapping RT CORBA priorities to native operating system priorities.</td>
</tr>
</tbody>
</table>

Valid values for `mapping_type` are:

- `direct` *(default)*—RT CORBA priorities are passed directly through as native priorities. The entire range of RT CORBA priorities may not be usable.
- `linear`—The entire RT CORBA priority range is mapped to the entire native priority range.
• continuous—RT CORBA priorities are mapped onto native priorities based on the minimum native value. For example, if the minimum native value is 8 and the CORBA priority is 22 then the mapped priority would be 30.

See 9.5.1 for more information on TAO’s priority mappings.

Table 9-2 Network Priority Mapping Selection Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBNetworkPriorityMapping mapping_type</td>
<td>Selects the algorithm to use when mapping RT CORBA priorities to network priorities.</td>
</tr>
</tbody>
</table>

Currently, the only valid value for mapping_type is:

• linear (default)—RT CORBA priorities are mapped to a series of network priorities, represented as diffserv codepoints. The linear mapping is the only network priority mapping provided with TAO.

See 9.5.2 for more information on enabling network priorities in TAO.

Table 9-3 Scheduling Policy Selection Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBSchedPolicy sched_policy</td>
<td>Specifies the scheduling policy used when mapping RT CORBA priorities to native priorities.</td>
</tr>
</tbody>
</table>

On some operating systems, the choice of scheduling policy affects how the priority mapping computations are performed. Each scheduling policy may have different low and high priority values, and therefore would affect the priority at which threads may run. Valid values for sched_policy are:

• SCHED_OTHER (default)—System-dependent default scheduling policy.
• SCHED_FIFO—FIFO scheduling policy, wherein the highest priority thread that can run is scheduled first.
• SCHED RR—Round-robin scheduling policy, wherein a fixed time-slice is provided for each thread.

You may need “super-user” or “Administrator” privileges to affect the scheduling policy.
On some operating systems, the choice of scheduling contention scope affects the preemption and execution of the threads allocated to RT CORBA thread pools. Valid values for \texttt{scope\_policy} are:

- \texttt{PROCESS} (default)—Threads compete for resources only with other threads in the same process.
- \texttt{SYSTEM}—Threads with system scheduling contention scope compete for resources against all threads in the system. This may not be available for all threading implementations.

### 9.5.7.2 RT Collocation Resolver

TAO normally optimizes collocated invocations (where the client and the target object are in the same address space). The effect of the ORB’s default collocation optimization is such that the client thread is used to carry out the request. As described in 17.4.5, this effect may be undesirable in real-time applications, because the client thread may not be running at the priority at which the request should be processed, possibly leading to priority inversions.

Therefore, TAO’s implementation of RT CORBA employs a special “real-time collocation resolver” (\texttt{RT\_Collocation\_Resolver}) to determine whether an invocation should be subject to collocation optimization. As described in 19.8.8, the RT collocation resolver considers factors other than just whether the request target object is in the same address space as the client when deciding if the collocation optimization should be applied.

The default behavior of TAO’s RT collocation resolver is appropriate for most real-time CORBA applications. However, its behavior can be disabled if the default (non-RT) collocation optimization resolution mechanism is desired.
The RT collocation resolver can be disabled using the -ORBDisableRTCollocation ORB initialization option.

### Table 9-5 RT Collocation Resolver Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBDisableRTCollocation {0</td>
<td>1}</td>
</tr>
</tbody>
</table>

See 19.8.8 for more information on this option.

### 9.6 Client-Propagated Priority Model

In the client-propagated priority model, a CORBA request is executed at the priority specified by the client when the request is invoked. The CORBA priority of the request is carried with the invocation. In this model, the server is obligated to execute the servant code that handles the request in a thread running at the appropriate native priority, according to the selected priority mapping. The client’s requested priority is carried to the server in a CORBA priority service context, and is passed back to the client from the server, along with the reply, through the service context.

#### 9.6.1 Specifying the Client-Propagated Priority Model in the Server

A server specifies that it supports the client-propagated priority model by setting the RTCORBA::CLIENT_PROPAGATED policy during POA creation. The policy is then exported to clients via object references generated through that POA.

Here, we extend the Messenger example from Chapter 3 to set the RTCORBA::CLIENT_PROPAGATED priority model policy in the Messenger server:

```cpp
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main (int argc, char* argv[]) { 
  try 
```
9.6 Client-Propagated Priority Model

```cpp
{ // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get a reference to Root POA and activate it.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::narrow (obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Get the RTORB.
    obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::narrow (obj.in());

    // Use the RTORB to create the CLIENT_PROPAGATED priority model policy.
    CORBA::PolicyList policies;
    policies.length(1);
    RTCORBA::Priority default_server_priority = 30;
    policies[0] = rtorb->create_priority_model_policy(
        RTCORBA::CLIENT_PROPAGATED,
        default_server_priority); // priority to use if not propagated from client

    // Create a child POA with CLIENT_PROPAGATED priority model in effect.
    PortableServer::POA_var child_poa =
        poa->create_POA ("MessengerPOA", mgr.in(), policies);

    // Create a Messenger_i servant.
    Messenger_i messenger_servant;

    // Register the servant with the new POA, obtain its object reference,
    // stringifying it, and write it to a file
    PortableServer::ObjectId_var oid =
        child_poa->activate_object (&messenger_servant);
    CORBA::Object_var messenger_obj = child_poa->id_to_reference (oid.in());
    CORBA::String_var str = orb->object_to_string (messenger_obj.in());
    std::ofstream iorFile ("Messenger.ior");
    iorFile << str.in() << std::endl;
    iorFile.close();
    std::cout << "IOR written to file Messenger.ior" << std::endl;

    // Accept requests from clients.
    orb->run();

    // Release resources.
    orb->destroy();
}
```

```
catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
}
```
return 0;
}

### 9.6.2 Using the Client-Propagated Priority Model in the Client

To use the client-propagated priority model in the client, we first check to see if the object reference we obtain from the server is configured with the `RTCORBA::CLIENT_PROPAGATED` policy. We do this by calling the `_get_policy()` operation on the object reference, narrowing the resulting policy object to `RTCORBA::PriorityModelPolicy`, then testing it to see if its value is `RTCORBA::CLIENT_PROPAGATED`:

```cpp
// Get the Messenger object reference.
CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");
Messenger_var messenger = Messenger::_narrow(obj.in());

// Get the policy from the object reference.
CORBA::Policy_var policy =
    messenger->_get_policy(RTCORBA::PRIORITY_MODEL_POLICY_TYPE);

// Check to see if it is of type RTCORBA::PriorityModelPolicy.
RTCORBA::PriorityModelPolicy_var priority_policy =
    RTCORBA::PriorityModelPolicy::narrow(policy.in());
if (CORBA::is_nil (priority_policy.in())) {
    std::cerr << "Messenger object does not support RTCORBA::PriorityModelPolicy"
             << std::endl;
    return 1;
}

// Check to see if the priority model is RTCORBA::CLIENT_PROPAGATED.
RTCORBA::PriorityModel priority_model = priority_policy->priority_model();
if (priority_model != RTCORBA::CLIENT_PROPAGATED) {
    std::cerr << "Messenger object does not support RTCORBA::CLIENT_PROPAGATED"
             << std::endl;
    return 1;
}

Next, we use the `RTCORBA::Current` interface to set the priority of the calling thread to the priority we want propagated to the server with the request. We call `resolve_initial_references("RTCurrent")` on the ORB to obtain the `RTCORBA::Current`, then use the `RTCORBA::Current`
object to set the priority of the current thread to the CORBA priority at which we want the request to be processed by the server:

```cpp
CORBA::Object_var current_obj = orb->resolve_initial_references("RTCurrent");
RTCORBA::Current_var current = RTCORBA::Current::_narrow(current_obj.in());
RTCORBA::Priority desired_priority = 10;
current->the_priority(desired_priority);
```

Now, when we invoke an operation on the Messenger object reference, the priority will be carried with the request to the server’s ORB, where it will be used to set the priority of the thread that processes the request:

```cpp
CORBA::String_var message = CORBA::string_dup("Howdy!");
messenger->send_message("TAO User", "Test", message.inout());
```

## 9.7 Server-Declared Priority Model

In the server-declared priority model, a CORBA request is executed at the priority specified by the server as the default for CORBA objects managed by the POA to which the policy applies, or at the priority specified on a per-object-reference basis. The server-declared model is appropriate when some operations must always be invoked at the same priority, regardless of the client thread making the request. In this model, the priority at which invocations on an object reference will be executed by the server is published in the object reference, where it is available to the client-side ORB. The client’s ORB can use this priority information to, for example, set the priority of the calling thread or choose the appropriate connection to the server over which to send the request. The priority is not carried with the request through the service context list as it is with the client-propagated priority model.

### 9.7.1 Specifying the Server-Declared Priority Model in the Server

The server specifies that it supports the server-declared priority model by setting the `RTCORBA::SERVER_DECLARED` policy, and the default priority value to use for executing requests, during POA creation. By default, operation invocations on CORBA objects managed by a particular POA will be executed at the default priority value specified at that POA’s creation. The server-declared policy and the priority value are exported to clients via object
references generated through that POA. The priority can also be set on a per-object-reference basis, as explained later in this section.

Here, we extend the Messenger example from Chapter 3 to set the RTCORBA::SERVER_DECLARED priority model policy in the Messenger server:

```cpp
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main (int argc, char* argv[]) {
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get a reference to Root POA and activate it.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Get the RTORB.
    obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());

    // Use the RTORB to create the SERVER_DECLARED priority model policy.
    CORBA::PolicyList policies;
    policies.length(1);
    RTCORBA::Priority default_server_priority = 30;
    policies[0] = rtorb->create_priority_model_policy (RTCORBA::SERVER_DECLARED,
                                                   default_server_priority);  // default priority to use

    // Create a child POA with SERVER_DECLARED priority model in effect.
    PortableServer::POA_var child_poa =
      poa->create_POA ("MessengerPOA", mgr.in(), policies);

    // Create a Messenger_i servant.
    Messenger_i messenger_servant;

    // Activate the Messenger in the new POA.
    PortableServer::ObjectId_var oid =
      child_poa->activate_object (&messenger_servant);

    // Obtain the Messenger object reference, stringify it, and write it to a file
    CORBA::Object_var messenger_obj = child_poa->id_to_reference (oid.in());
    CORBA::String_var str = orb->object_to_string (messenger_obj.in());
  }
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
You can use operations defined on the RTPortableServer::POA interface to override the POA’s default priority on a per-object-reference basis. For example, we can use the activate_object_with_priority() function rather than activate_object() to activate the Messenger object with a CORBA priority other than the default, specified when the POA was created:

```cpp
// Use the POA as a RT POA.
RTPortableServer::POA_var rt_poa =
   RTPortableServer::POA::_narrow (child_poa.in());

// Activate the Messenger in the new POA with a specific priority.
RTCORBA::Priority messenger_specific_priority = 50;
PortableServer::ObjectId_var oid =
   rt_poa->activate_object_with_priority (  
      &messenger_servant,  
      messenger_specific_priority);
```

Other operations on the RTPortableServer::POA interface are used similarly, including:

- create_reference_with_priority(),
- create_reference_with_id_and_priority(), and
- activate_object_with_id_and_priority(),

depending upon the POA policies (e.g., PortableServer::USER_ID or PortableServer::SYSTEM_ID) in effect. If the POA does not support the RTPortableServer::SERVER_DECLARED priority policy, and you try to
use one of the above operations to set the priority of the object reference, the
PortableServer::WrongPolicy exception is raised.

9.7.2 Using the Server-Declared Priority Model in the Client
You do not need to do anything special on the client-side to use an object
reference with the server-declared priority model. The server automatically
executes operations invoked on that object reference at the priority declared in
the object reference. However, clients can use the priority information in the
object reference to make decisions about how (or whether) to invoke the
request. In the following example, our MessengerClient will invoke the
send_message() operation only if the Messenger object reference supports
the server-declared priority model, and the priority value is sufficiently high:

```c++
// Get the Messenger object reference.
CORBA::Object_var obj = orb->string_to_object ("file://Messenger.ior");
Messenger_var messenger = Messenger::_narrow (obj.in());

// Get the policy from the object reference.
CORBA::Policy_var policy =
    messenger->_get_policy (RTCORBA::PRIORITY_MODEL_POLICY_TYPE);

// Check to see if it is of type RTCORBA::PriorityModelPolicy.
RTCORBA::PriorityModelPolicy_var priority_policy =
    RTCORBA::PriorityModelPolicy::_narrow (policy.in ());
if (CORBA::is_nil (priority_policy.in ())) {
    std::cerr << "Messenger object does not support RTCORBA::PriorityModelPolicy"
              << std::endl;
    return 1;
}

// Check to see if the priority model is RTCORBA::SERVER_DECLARED.
RTCORBA::PriorityModel priority_model = priority_policy->priority_model();
if (priority_model != RTCORBA::SERVER_DECLARED) {
    std::cerr << "Messenger object does not support RTCORBA::SERVER_DECLARED"
              << std::endl;
    return 1;
}

// Check to see if the server’s priority value is high enough for us.
RTCORBA::Priority desired_priority = 30;
RTCORBA::Priority server_priority = priority_policy->server_priority();
if (server_priority < desired_priority) {
    std::cerr << "Messenger object’s priority is too low." << std::endl;
    return 1;
}
```
9.8 Using the RTScheduling::Current

The RTScheduling::Current exists to manage distributable threads. Through its interface, an application is able to spawn new distributable threads, and begin, update, and end scheduling segments. The Current object provides a thread with an identity as well as a way to obtain references to other distributable threads. The interface is described in detail in 9.4.3.

9.8.1 Spawning New Distributable Threads

The Current’s spawn() operation will create a new distributable thread, to invoke the do() operation of the supplied RTScheduling::ThreadAction reference. Spawning a new distributable thread has the side effect of creating a new operating system thread bound to the current process. The spawn() operation allows you to set the priority of the new thread and supply a scheduling parameter appropriate for whatever scheduling discipline was chosen for the application. TAO’s implementation of spawn() implicitly associates the newly created distributable thread with a scheduling segment by invoking begin_scheduling_segment() right before calling the thread function, then invoking end_scheduling_segment() immediately after.

The following code fragment, found in $TAO_ROOT/tests/RTScheduling/DT_spawn/test.cpp, shows how to use the RTScheduling::Current::spawn() operation.

The application begins as any other CORBA application, setting up some pointers that will refer to various objects, then calling CORBA::ORB_init(). Note that, in this case, the sched_param and implicit_sched_param values are initialized to nil policy references. In this example, the particular scheduler we are using does not support any particular scheduling parameters.

#include "../Scheduler.h"   // for class TAO_Scheduler
#include "Thread_Action.h"  // for class Test_Thread_Action
#include "tao/RTScheduling/RTScheduler_Manager.h"
int main (int argc, char* argv [])
{
    CORBA::ORB_var orb;
    RTScheduling::Current_var current;
    const char * name = 0;
    CORBA::Policy_var sched_param = CORBA::Policy::_nil();
    CORBA::Policy_var implicit_sched_param = CORBA::Policy::_nil();
    Test_Thread_Action thread_action;
    try {
        orb = CORBA::ORB_init (argc, argv);
        A distributable thread cannot exist without a scheduler, so the next portion of
        the code shows the initialization of a scheduler. TAO deviates a bit from the
        RT CORBA specification by supplying a scheduler manager, through which
        an externally created scheduler may be associated with the ORB. The
        scheduler used for this example is trivial; its implementation is available in the
        parent directory for this example, in
        $TAO_ROOT/tests/RTScheduling/Scheduler.h.
        CORBA::Object_var manager_obj =
            orb->resolve_initial_references ("RTSchedulerManager");
        TAO_RTScheduler_Manager_var manager =
            TAO_RTScheduler_Manager::_narrow (manager_obj.in());
        TAO_Scheduler* scheduler;
        ACE_NEW_RETURN (scheduler, TAO_Scheduler (orb.in ()), -1);
        manager->rtscheduler (scheduler);
        Now that we have created a scheduler, we are ready to obtain a scheduling
        current and use it to spawn a distributable thread.
        CORBA::Object_var current_obj =
            orb->resolve_initial_references ("RTScheduler_Current");
        current = RTScheduling::Current::_narrow (current_obj.in());
        try {
            ACE_DEBUG ((LM_DEBUG,
                ACE_TEXT("Invoking DT spawn without calling "
                    "begin_scheduling_segment...\n")));
            ACE_CString data ("Harry Potter");
9.8 Using the RTScheduling::Current

```c
char* thread_data = ACE_const_cast(char *, data.c_str());
CORBA::ULong stack_size = 0;
RTCORBA::Priority base_priority = 0;
current->spawn (&thread_action,
    thread_data,
    name,
    sched_param.in(),
    implicit_sched_param.in(),
    stack_size,
    base_priority);
```

At this point, the new thread is off and running. Notice that the `spawn()` operation takes several arguments:

- **thread_action**—a reference to an object that derives from `RTScheduler::ThreadAction`. The `do()` operation of this object is invoked as the interesting part of the thread function. See 9.4.2 for more information on the `ThreadAction` interface. In this example, the class `Test_Thread_Action` is defined in `$TAO_ROOT/tests/RTScheduling/DT_spawn/Thread_Action.h`.

- **thread_data**—the argument that is passed to the thread function. In this case, the thread function takes a string as an argument.

- **name**—the identity supplied to the new distributable thread. As a side-effect, this identity is supplied to `begin_scheduling_segment()` as part of the thread function. In this example, the name is empty since no other distributable threads exist when `spawn()` is called.

- **sched_param**—depending on the scheduling discipline applied to the application, this argument may point to information required to schedule this thread. It is forwarded to `begin_scheduling_segment()`. In this example, the trivial scheduler does not require any parameters, thus `sched_param` is nil.

- **implicit_sched_param**—also passed to `begin_scheduling_segment()`, this argument is used to update the scheduling parameter that would be used if one were needed and `sched_param` was nil. In this example, scheduling parameters are not needed, so this value is also nil.

- **stack_size**—the next argument supplied is the size of the stack for the newly spawned thread. A non-zero value would indicate the desired stack size.
size; passing zero, as in this example, indicates the operating system
defined default stack size should be used.

- **base_priority**—the last argument supplied is the base priority value
  for the newly spawned thread. In this example, multiple thread priorities
  are not being used, so the base priority value is passed as zero.

The rest of the example in the distribution shows how to manage distributable
threads using the current.

### 9.8.2 Managing Scheduling Segments

*Scheduling segments* are logical entities that may cross application
boundaries. They are used to define the life span of a distributable thread.
Depending upon the scheduling discipline used, a scheduling segment may
have information associated with it, such as a completion deadline, that may
cause the priority of the associated thread to be dynamically changed.
Scheduling segments are used by the scheduler to associate requests or
activities with application threads of appropriate priority. Scheduling
segments have a beginning and an end; the end need not be in the same
process as the beginning. Segments may be nested or consecutive. A
distributable thread may span many scheduling segments. Scheduling
segments may also be updated, changing the associated schedule parameter,
and possibly raising or lowering the priority.

The following example demonstrates the management of scheduling
segments. This example is found in
`$TAO_ROOT/tests/RTScheduling/Current/Thread_Task.cpp`. Here,
we look at `Thread_Task::svc()`, which provides an implementation of a
thread function just like any class derived from `ACE_Task_Base`. The code
below begins three nested scheduling segments, uses the current to obtain a
list of the currently scheduled segments, then finally ends each segment.

```c++
int Thread_Task::svc (void)
{
  try {
    const char * name = 0;
    CORBA::Policy_var sched_param = CORBA::Policy::_nil();
    CORBA::Policy_var implicit_sched_param = CORBA::Policy::_nil();

    this->current_->_begin_scheduling_segment ("Fellowship of the Ring",
                                              sched_param.in(),
                                              implicit_sched_param.in());
```
size_t count = 0;
ACE_OS::memcpy(
  &count,
  this->current_->id()->get_buffer(),
  this->current_->id()->length());

ACE_DEBUG((LM_DEBUG,
  ACE_TEXT("Starting Distributable Thread \%d with 3 nested scheduling "
    "segments...\n"),
    count));

// Start - Nested Scheduling Segment
this->current_->begin_scheduling_segment(
  "Two Towers",
  sched_param.in(),
  implicit_sched_param.in());

// Start - Nested Scheduling Segment
this->current_->begin_scheduling_segment(
  "The Return of the King",
  sched_param.in(),
  implicit_sched_param.in());

RTScheduling::Current::NameList* segment_name_list =
  this->current_->current_scheduling_segment_names();
{
  ACE_GUARD_RETURN(TAO_SYNCH_MUTEX, ace_mon, *lock_, -1);
  ACE_DEBUG((LM_DEBUG,
    ACE_TEXT("Segment Names for DT \%d :\n"),
    count));

  for (unsigned int i = 0; i < segment_name_list->length(); ++i)
    {
    ACE_DEBUG((LM_DEBUG,
      ACE_TEXT("\%s\n"),
      (*segment_name_list)[i].in()));
  }
}

// End - Nested Scheduling Segment
this->current_->end_scheduling_segment(name);

// End - Nested Scheduling Segment
this->current_->end_scheduling_segment(name);

// End - Nested Scheduling Segment
this->current_->end_scheduling_segment(name);
Real-Time CORBA

ACE_DEBUG ((LM_DEBUG, 
    ACE_TEXT("DT %d terminated ...
"), 
    count));

    
    ACE_GUARD_RETURN (TAO_SYNCH_MUTEX, ace_mon, *shutdown_lock_, -1); 
    --active_thread_count_;
    if (active_thread_count_ == 0)
    {
        // Without this sleep, we will occasionally get BAD_INV_ORDER 
        // exceptions on fast dual processor machines.
        ACE_OS::sleep (1);
        orb_->shutdown (0);
    }
}
} 

9.9 Real-Time CORBA Examples

Throughout this chapter we have shown several code fragments illustrating 
the use of real-time CORBA features. Many of the examples we have shown 
are available as executable code distributed with TAO. This code is found in 
the following directories:

- $TAO_ROOT/DevGuideExamples/RTCORBA—This directory contains a 
  variation of the Messenger application introduced in Chapter 3. This 
  example uses the RTORB in the client to create a private connection 
  policy and uses the RTCORBA::Current for setting the priority to be 
  consistent with the server. On the server side, the example shows the use 
  of a thread pool with some number of lanes (the number depending on the 
  native platform).

- $TAO_ROOT/examples/RTCORBA/Activity—The example in this 
  directory highlights configuration of RTORB and RTPOA characteristics 
  including the priority mapping policy.
9.9 Real-Time CORBA Examples

- `$TAO_ROOT/examples/RTScheduling`—This directory contains several applications that demonstrate the use of distributable threads and various scheduling disciplines.

- `$TAO_ROOT/tests/RTCORBA`—This directory is the top level directory for a suite of tests that are run regularly to ensure the integrity of TAO’s real-time CORBA implementation. These tests are focused on isolating particular features, but do provide insight into use of RT CORBA mutexes, priority-banded connections, client-propagated and server-declared priority models, collocation, and more.

- `$TAO_ROOT/tests/RTScheduling`—This directory includes a suite of tests focused on the use of the real-time scheduler. Specific tests highlight using distributable threads, canceling threads, and the scheduling current.
10.1 Introduction

Portable Interceptors in CORBA are objects that the ORB invokes at predefined points in the request and reply paths of an operation invocation (request interceptors) or during the generation of an IOR (IOR interceptors). As an application developer, you define the code executed in an interceptor. Since interceptors exist orthogonally to the operation invocations they monitor/modify, interceptor code can be added without affecting existing client and server code. Portable Interceptors can perform a variety of information collection and authentication tasks, including the following:

- Gathering debugging information about messages sent between clients and servers.
- Logging usage or access information about distributed objects.
- Performing security and authenticity checks in a distributed system.

TAO supports both request interceptors and IOR interceptors. Except for minor differences, TAO’s implementation conforms to the CORBA 3.0.3 Portable Interceptor specification (Chapter 21 of OMG Document).
Portable Interceptors

Portable interceptor support in TAO is controlled by the `interceptors` build flag. By default, interceptors are enabled unless `minimum_corba=1` or `interceptors=0`. For more information on build flags, refer to 2.5.


10.2 Using TAO Request Interceptors

TAO request interceptors can be attached at four points along the request/reply path of client and server communications. On the client, they can be activated when a request is sent or when a reply is received. On the server, they can be activated when a target operation is called, or when the reply is sent. There are ten interception operations, discussed in the next four sections. Figure 10-1 shows the relationship of clients and servers to the ten interception operations.

![Figure 10-1 Client and Server Interception Operations](image)

10.2.1 Client Request Interceptors

Client request interceptors implement the following interfaces, which are defined in `$TAO_ROOT/tao/PortableInterceptor.pidl`:

```plaintext
Client Request Interceptor
- send_request
- send_poll
- receive_reply
- receive_exception
- receive_other

Server Request Interceptor
- receive_request
- receive_request_service_contexts
- send_reply
- send_exception
- send_other
```
module PortableInterceptor
{
    local interface Interceptor
    {
        readonly attribute string name;
        void destroy ();
    };

    local interface ClientRequestInterceptor : Interceptor
    {
        void send_request (in ClientRequestInfo ri) raises (ForwardRequest);
        void send_poll (in ClientRequestInfo ri);
        void receive_reply (in ClientRequestInfo ri);
        void receive_exception (in ClientRequestInfo ri) raises (ForwardRequest);
        void receive_other (in ClientRequestInfo ri) raises (ForwardRequest);
    };

    // additional interfaces omitted for brevity
};

To use interceptors on the client-side, developers define a new class that inherits from PortableInterceptor::ClientRequestInterceptor, and implement the five operations that correspond to the client side interception points, plus operations that provide the name of the interceptor and destroy the interceptor.

TAO does not support the send_poll() client interception point. The send_poll() operation is specific to time-independent invocations and TAO does not currently support time-independent invocations.

10.2.1.1 Client Interception Points

The four client interception points available in TAO are send_request(), receive_reply(), receive_exception(), and receive_other(). The send_request() interception point allows an interceptor to monitor or change the service context before a request is sent to the server. The receive_reply() point intercepts a reply after it has returned from the server but before it has been passed to the client. The receive_exception() point is invoked when an exception occurs, before the exception is raised to the client. The receive_other() interception point allows the interceptor to monitor responses that are neither normal replies nor exceptions. An example of this would be a LOCATION_FORWARD response.
10.2.2 Server Request Interceptors

Interceptors on the server side implement the following interfaces, which are defined in $TAO_ROOT/tao/PortableInterceptor.pidl:

```cpp
module PortableInterceptor {

    local interface Interceptor {
        readonly attribute string name;
        void destroy ();
    }

    local interface ServerRequestInterceptor : Interceptor {
        void receive_request_service_contexts (in ServerRequestInfo ri)
            raises (ForwardRequest);
        void receive_request (in ServerRequestInfo ri) raises (ForwardRequest);
        void send_reply (in ServerRequestInfo ri);
        void send_exception (in ServerRequestInfo ri) raises (ForwardRequest);
        void send_other (in ServerRequestInfo ri) raises (ForwardRequest);
    }

    // additional interfaces omitted for brevity.
};
```

To use interceptors on the server side, developers define a new class that inherits from `PortableInterceptor::ServerRequestInterceptor`, and implement the five operations that correspond to the five server side interception points, plus operations that provide the name of the interceptor and destroy the interceptor.

10.2.2.1 Server Interception Points

The `receive_request_service_context()` interception point is called before the servant manager is called. The operation parameters that are passed as part of the request’s service context are not available to the interceptor at this point. Any service context information the interceptor needs must be obtained from the `request scope` `PICurrent`. The `PICurrent` is explained in 10.5. The `receive_request()` point allows the interceptor to monitor request information once all operation parameters are available. An interceptor that implements the `send_reply()` operation can monitor and modify the reply service context after the server operation has been invoked but before the reply is sent to the client. The `send_exception()` point allows the interceptor to inspect exception information and modify the reply service context before the exception is sent to the client. Interceptors implementing
the send_other() operation can inspect the information available when the request results in something other than a normal reply or an exception.

### 10.2.3 Request Parameters

Request interceptors access request information through ClientRequestInfo and ServerRequestInfo objects, which are given as parameters to their respective interception points. Client and server RequestInfo objects inherit from a common interface defined in $TAO_ROOT/tao/PortableInterceptor.pidl:

```plaintext
local interface RequestInfo
{
    readonly attribute unsigned long request_id;
    readonly attribute string operation;

    readonly attribute Dynamic::ParameterList arguments;
    readonly attribute Dynamic::ExceptionList exceptions;
    readonly attribute Dynamic::ContextList contexts;
    readonly attribute Dynamic::RequestContext operation_context;

    readonly attribute any result;
    readonly attribute boolean response_expected;
    readonly attribute Messaging::SyncScope sync_scope;

    readonly attribute ReplyStatus reply_status;
    readonly attribute Object forward_reference;

    any get_slot (in SlotId id) raises (InvalidSlot);
    IOP::ServiceContext get_request_service_context (in IOP::ServiceId id);
    IOP::ServiceContext get_reply_service_context (in IOP::ServiceId id);
};
```

The ClientRequestInfo interface extends the RequestInfo interface with attributes and operations of interest to client-side interceptors.

```plaintext
local interface ClientRequestInfo : RequestInfo
{
    readonly attribute Object target;
    readonly attribute Object effective_target;
    readonly attribute IOP::TaggedProfile effective_profile;

    readonly attribute any received_exception;
    readonly attribute CORBA::RepositoryId received_exception_id;

    IOP::TaggedComponent get_effective_component (in IOP::ComponentId id);
    IOP::TaggedComponentSeq get_effective_components (in IOP::ComponentId id);
};
```
The `ServerRequestInfo` interface extends the `RequestInfo` interface with attributes and operations of interest to the server-side interceptors.

```cpp
local interface ServerRequestInfo : RequestInfo
{
    readonly attribute any sending_exception;
    readonly attribute CORBA::OctetSeq object_id;
    readonly attribute CORBA::OctetSeq adapter_id;
    readonly attribute CORBA::RepositoryId target_most_derived_interface;

    CORBA::Policy get_server_policy (in CORBA::PolicyType type);
    void set_slot (in SlotId id, in any data) raises (InvalidSlot);
    boolean target_is_a (in CORBA::RepositoryId id);
    void add_reply_service_context (in IOP::ServiceContext service_context,
                                    in boolean replace);
};
```

For an explanation of the attributes and operations of a `RequestInfo` object, as well as their applicability to each interception point, see Chapter 21 of the CORBA 3.0.3 specification.

### 10.2.4 Registering Interceptors

As a developer, you provide the code to register your application’s interceptors with the ORB. Interceptors are installed in the ORB with an `ORBInitializer` object and registered by implementing its `pre_init()` or `post_init()` method and calling `PortableInterceptor::register_orb_initializer()` prior to calling `CORBA::ORB_init()`. The specifics of Interceptor Initialization are shown in the example in the next section. Developers need to be aware that certain operations that need a pointer to the ORB can not be invoked during interceptor registration, because the registration occurs within the call to `ORB_init()`, and no ORB pointer exists yet.

To allow for this, the `ORBInitInfo` interface contains functions and attributes that hold the arguments passed to `ORB_init()`, a reference to the
10.2 Using TAO Request Interceptors

CodecFactory, and additional information that would otherwise be unavailable.

10.2.5 Example: A Simple Authentication Interceptor

Our first example uses interceptors to add a user name to each request that a client makes. The client’s interceptor provides the name that is sent with the request. The server’s interceptor authenticates the user before the request is dispatched to the servant. Our example extends the Messenger example from Chapter 3. The complete source code for this example is in the TAO source code distribution in the directory

$TAO_ROOT/DevGuideExamples/PortableInterceptors/Auth.

Note This example is not meant as a secure authentication solution. Please refer to the Chapter 29 for a more thorough treatment of the subject of security.

10.2.5.1 Messenger Interface

The definition of the Messenger interface has not changed:

```cpp
interface Messenger
{
    boolean send_message (in string user_name,
                     in string subject,
                     inout string message);
};
```

10.2.5.2 Messenger Implementation Class

We now define the `Messenger_i` implementation class as follows:

```cpp
#include "MessengerS.h"

class Messenger_i : public virtual POA_Messenger
{
 public:
     //Constructor
     Messenger_i (void);

     //Destructor
     virtual ~Messenger_i (void);

     virtual CORBA::Boolean send_message (}
**Portable Interceptors**

The implementation of the `Messenger_i` class is as follows:

```cpp
#include "Messenger_i.h"
#include <iostream>

// Implementation skeleton constructor
Messenger_i::Messenger_i (void)
{
}

// Implementation skeleton destructor
Messenger_i::~Messenger_i (void)
{
}

CORBA::Boolean Messenger_i::send_message (const char* user_name,
                                          const char* subject,
                                          char*& message)
throw (CORBA::SystemException)
{
  std::cout << "Message from: " << user_name << std::endl;
  std::cout << "Subject:      " << subject << std::endl;
  std::cout << "Message:      " << message << std::endl;
  CORBA::string_free (message);
  message = CORBA::string_dup ("Thanks for the message."");
  return 1;
}
```

### 10.2.5.3 Defining the Client Request Interceptor

The client-side interceptor is defined in the `ClientInterceptor` class. This class inherits from `PortableInterceptor::ClientRequestInterceptor`.

```cpp
#include <tao/PortableInterceptorC.h>

class ClientInterceptor : public PortableInterceptor::ClientRequestInterceptor
{
```
10.2 Using TAO Request Interceptors

public:
    ClientInterceptor (void);
    virtual ~ClientInterceptor () ;

    virtual char* name ()
        throw (CORBA::SystemException);

    virtual void destroy ()
        throw (CORBA::SystemException);

    virtual void send_poll (PortableInterceptor::ClientRequestInfo_ptr ri)
        throw (CORBA::SystemException);

    virtual void send_request (PortableInterceptor::ClientRequestInfo_ptr ri)
        throw (CORBA::SystemException,
           PortableInterceptor::ForwardRequest);

    virtual void receive_reply (PortableInterceptor::ClientRequestInfo_ptr ri)
        throw (CORBA::SystemException);

    virtual void receive_other (PortableInterceptor::ClientRequestInfo_ptr ri)
        throw (CORBA::SystemException,
           PortableInterceptor::ForwardRequest);

    virtual void receive_exception (PortableInterceptor::ClientRequestInfo_ptr ri)
        throw (CORBA::SystemException,
           PortableInterceptor::ForwardRequest);

private:
    const char *myname_ ;
};

In this example, we have overridden the operations that correspond to the five interception points as well as the name() and destroy() operations.

10.2.5.4 Implementing the Client Request Interceptor

Next, we implement the client request interceptor. The constructor initializes the name of our interceptor. The name() operation’s implementation is simple.

    ClientInterceptor::ClientInterceptor (void)
      : myname_ ("Client_Authentication_Interceptor")
    {
      std::cout << "Calling ClientInterceptor constructor." << std::endl;
    }

    char* ClientInterceptor::name ()
Portable Interceptors

throw (CORBA::SystemException)
{
    std::cout << "Calling ClientInterceptor name() method" << std::endl;
    return CORBA::string_dup (this->myname_);
}

To illustrate how information can be passed using the service context list, we insert some user name information into the service context list at the client’s send_request() interception point. Later, we will show how we can extract and use this as validation information on the server side at the server’s receive_request() interception point.

In our interceptor’s send_request() operation, we first log information about the request to standard output. Then, we insert information into the service context list. Recall that the service context list is a sequence of structures, each containing a context identifier of type unsigned long and a sequence of octets. You can use these fields to pass any information you want with the request. Here, we set the context_id field to an arbitrary value and populate the context_data field with the user’s name after we convert the name from a string to a sequence of octets.

const CORBA::ULong service_ctx_id = 0xdeed;

void ClientInterceptor::send_request (PortableInterceptor::ClientRequestInfo_ptr ri)
    throw (CORBA::SystemException,
           PortableInterceptor::ForwardRequest)
{
    std::cout << "Calling send_request()." << std::endl;

    IOP::ServiceContext sc;
    sc.context_id = service_ctx_id;

    const char user_name[] = "Ron Klein";
    std::cout << "User’s Name: " << user_name << std::endl;
    CORBA::ULong string_len = sizeof (user_name) + 1;
    CORBA::Octet *buf = new CORBA::Octet [string_len];
    ACE_OS::strcpy (ACE_reinterpret_cast (char *, buf), user_name);
    sc.context_data.replace (string_len, string_len, buf, 1);

    // Add this context to the service context list.
    ri->add_request_service_context (sc, 0);
}
Our `receive_reply()`, `receive_other()`, and `receive_exception()` operations do nothing of any importance. They simply log information about the request to standard output.

```cpp
void ClientInterceptor::receive_reply (PortableInterceptor::ClientRequestInfo_ptr ri)
throw (CORBA::SystemException)
{
    ACE_UNUSED_ARG(ri);
    std::cout << "Calling receive_reply()." << std::endl;
}

void ClientInterceptor::receive_other (PortableInterceptor::ClientRequestInfo_ptr)
throw (CORBA::SystemException,PortableInterceptor::ForwardRequest)
{
    std::cout << "Calling receive_other()." << std::endl;
}

void ClientInterceptor::receive_exception (PortableInterceptor::ClientRequestInfo_ptr ri)
throw (CORBA::SystemException,PortableInterceptor::ForwardRequest)
{
    ACE_UNUSED_ARG(ri);
    std::cout << "Calling receive_exception()." << std::endl;
}
```

### 10.2.5.5 Developing the Client and Installing the Interceptor

To use our client request interceptor, we implement the `post_init()` function of a `ClientInitializer` object, which inherits from `PortableInterceptor::ORBInitializer`.

```cpp
class ClientInitializer : public virtual PortableInterceptor::ORBInitializer
{
    virtual void post_init (PortableInterceptor::ORBInitInfo_ptr info)
    throw (CORBA::SystemException);
};
```

If we were registering another interceptor that needed access to this interceptor’s initial services, we would choose to register this interceptor in `pre_init()`. Since no other interceptors need access to this interceptor’s
services, we register this interceptor in `post_init()`. In contrast to `pre_init()`, `post_init()` is executed at the point in ORB initialization when all initial references are available. Our `post_init()` instantiates a `ClientRequestInterceptor` and registers it by calling `ORBInitInfo::add_client_request_interceptor()`.

```cpp
void ClientInitializer::post_init (PortableInterceptor::ORBInitInfo_ptr info)
throw (CORBA::SystemException)
{
    // Create and register the request interceptors.
    PortableInterceptor::ClientRequestInterceptor_var ci = new ClientInterceptor();
    info->add_client_request_interceptor (ci.in());
}
```

With the initializer written, we develop a client and install our interceptor by creating and registering the `ClientInitializer` object before calling `CORBA::ORB_init()`. We include header files for both the Messenger interface and our Messenger initializer class definitions and instantiate an `ORBInitializer_var` which is passed as a parameter to `PortableInterceptor::register_orb_initializer()`.

```cpp
#include "MessengerC.h"
#include "ClientInitializer.h"
#include <orbsvcs/CosNamingC.h>
#include <iostream>

int main (int argc, char* argv[]) {
    try {
        // Create and register our ORBInitializer.
        PortableInterceptor::ORBInitializer_ptr temp_initializer = PortableInterceptor::ORBInitializer::_nil();
        temp_initializer = new ClientInitializer;
        PortableInterceptor::ORBInitializer_var orb_initializer = temp_initializer;
        PortableInterceptor::register_orb_initializer (orb_initializer.in());

        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "Client ORB");
```
Next, we acquire a reference to the Messenger object and use it to send a message. The user name authentication is added transparently by the interceptor and passed through the service context with the rest of the request when `send_message()` is invoked.

```cpp
// Read and destringify the Messenger object's IOR.
CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");
if (CORBA::is_nil(obj.in())) {
    std::cerr << "Could not get Messenger IOR." << std::endl;
    return 1;
}

// Narrow the IOR to a Messenger object reference.
Messenger_var messenger = Messenger::_narrow(obj.in());
if (CORBA::is_nil(messenger.in())) {
    std::cerr << "IOR was not a Messenger object reference." << std::endl;
    return 1;
}

// Send a message the the Messenger object.
CORBA::String_var message = CORBA::string_dup("Hello!");
messenger->send_message("TAO User", "TAO Test", message.inout());

// Print the Messenger's reply.
std::cout << "Reply: " << message.in() << std::endl;
}
catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
    return 1;
}

return 0;
}
```

### 10.2.5.6 Defining the Server Request Interceptor

The server-side interceptor is defined in the `ServerRequestInterceptor` class. This class inherits from the `ServerInterceptor::ServerRequestInterceptor` class.

```cpp
#include "tao/PortableInterceptorC.h"

class ServerInterceptor : public PortableInterceptor::ServerRequestInterceptor {

    public:
        ServerInterceptor (void);
        ~ServerInterceptor () ;

```
virtual char* name ()
    throw (CORBA::SystemException);

virtual void destroy ()
    throw (CORBA::SystemException);

virtual void receive_request (PortableInterceptor::ServerRequestInfo_ptr ri)
    throw (CORBA::SystemException,
          PortableInterceptor::ForwardRequest,
          CORBA::NO_PERMISSION);

virtual void receive_request_service_contexts (PortableInterceptor::ServerRequestInfo_ptr ri)
    throw (CORBA::SystemException,
          PortableInterceptor::ForwardRequest);

virtual void send_reply (PortableInterceptor::ServerRequestInfo_ptr ri)
    throw (CORBA::SystemException);

virtual void send_exception (PortableInterceptor::ServerRequestInfo_ptr ri)
    throw (CORBA::SystemException,
          PortableInterceptor::ForwardRequest);

virtual void send_other (PortableInterceptor::ServerRequestInfo_ptr ri)
    throw (CORBA::SystemException,
          PortableInterceptor::ForwardRequest);

private:
    const char *myname_;
char* ServerInterceptor::name ()
    throw (CORBA::SystemException)
{
    std::cout << "Calling ServerInterceptor name() method" << std::endl;
    return CORBA::string_dup (this->myname_);
}

Recall that the client request interceptor passes information along with the request in the service context list. We now want to extract this information on the server side at the server’s receive_request() interception point.

In our interceptor’s receive_request() operation, we call target_is_a() to verify that the remote invocation is requesting a Messenger object. Then we obtain the service context data, which contains the user name as an octet sequence, and cast it to a char * so that it can be compared to allowed_users[]. If the user name matches and element in allowed_users[], authentication is successful.

const IOP::ServiceId service_id = 0xdeed;
const unsigned int num_allowed_users = 4;
const static char* allowed_users[num_allowed_users+1] =
    {"Ron Klein", "Scott Case", "Mark Hodge", "Greg Black", 0};
const char* restricted_interfaces[1] = {"IDL:Messenger:1.0"};

void ServerInterceptor::receive_request (PortableInterceptor::ServerRequestInfo_ptr ri)
    throw (CORBA::SystemException,
           PortableInterceptor::ForwardRequest,
           CORBA::NO_PERMISSION)
{
    bool permission_granted = false;
    std::cout << "Calling receive_request()." << std::endl;

    if (ri->target_is_a(restricted_interfaces[0]))
    {
        IOP::ServiceId id = service_id;

        // Check that the request service context can be retrieved.
        IOP::ServiceContext_var sc = ri->get_request_service_context (id);
        class IOP::ServiceContext::_tao_seq_Octet ocSeq = sc->context_data;
        const char * buf =
            ACE_reinterpret_cast (const char *, ocSeq.get_buffer());

        for (unsigned int i=0; i<num_allowed_users; ++i) 
            { 
                if (ACE_OS::strcmp (buf, allowed_users[i]) == 0) 
                    { permission_granted = true; break; } 
            }
    }
permission_granted = true;
break;
}
}

if (permission_granted == true) {
    std::cout << "Permission Granted " << std::endl;
} else {
    std::cout << "Permission Denied " << std::endl;
    throw CORBA::NO_PERMISSION();
}

Our other interception point operations do nothing important. They log information about the request to standard output.

void ServerInterceptor::receive_request_service_contexts (  
    PortableInterceptor::ServerRequestInfo_ptr ri)  
throw (CORBA::SystemException,  
    PortableInterceptor::ForwardRequest)  
{
    ACE_UNUSED_ARG(ri);  
    std::cout << "Calling receive_request_service_contexts()." << std::endl;
}

void ServerInterceptor::send_reply (  
    PortableInterceptor::ServerRequestInfo_ptr ri)  
throw (CORBA::SystemException)  
{
    ACE_UNUSED_ARG(ri);  
    std::cout << "Calling send_reply()." << std::endl;
}

void ServerInterceptor::send_exception (  
    PortableInterceptor::ServerRequestInfo_ptr ri)  
throw (CORBA::SystemException,  
    PortableInterceptor::ForwardRequest)  
{
    ACE_UNUSED_ARG(ri);  
    std::cout << "Calling send_exception()." << std::endl;
}

void ServerInterceptor::send_other (  
    PortableInterceptor::ServerRequestInfo_ptr ri)  
throw (CORBA::SystemException,  
    PortableInterceptor::ForwardRequest)  
{
10.2 Using TAO Request Interceptors

10.2.5.8 Developing the Server and Installing the Interceptor

We develop a server and install our interceptor by registering it with the ORB. The procedure is very similar to installation of the client interceptor. We are extending the MessengerServer example from Chapter 3. Few changes to the original code are required to use interceptors and we will explain them as we go along.

The code for the server interceptor initializer is similar to its client counterpart. The ServerInitializer class inherits from PortableInterceptor::ORBInitializer, and we implement the post_init() operation by creating a ServerRequestInterceptor and using it as a parameter to ORBInitInfo::add_server_request_interceptor().

```cpp
void ServerInitializer::post_init (  
    PortableInterceptor::ORBInitInfo_ptr info)  
    throw (CORBA::SystemException)  
{  
    // Create and register the request interceptors.  
    PortableInterceptor::ServerRequestInterceptor_var si = new ServerInterceptor();  
    info->add_server_request_interceptor (si.in());  
}
```

In the server, we include header files for both the Messenger interface and our Messenger interceptor initializer class definitions. We then instantiate a new ServerInitializer object and register the server interceptor with a call to PortableInterceptor::ORBInitializer(). Then we initialize the ORB as usual.

```cpp
#include "Messenger_i.h"  
#include "MessengerS.h"  
#include "ServerInitializer.h"  
#include <iostream>  
#include <fstream>  

int main (int argc, char* argv[])  
{  
    try {  
```
Portable Interceptors

// Create and register our ORBInitializer.
ServerInitializer* temp_initializer = new ServerInitializer;
PortableInterceptor::ORBInitializer_var orb_initializer =
temp_initializer;
PortableInterceptor::register_orb_initializer (orb_initializer.in());

// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "Server ORB");

The next part of the server is unchanged from the MessengerServer in
Chapter 3.

// Get reference to the RootPOA.
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

// Activate the POAManager.
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Create a servant.
Messenger_i messenger_servant;

// Register the servant with the RootPOA, obtain its object
// reference, stringify it, and write it to a file.
PortableServer::ObjectId_var oid =
    poa->activate_object(&messenger_servant);
CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
CORBA::String_var str = orb->object_to_string(messenger_obj.in());
std::ofstream iorFile("Messenger.ior");
iorFile << str.in() << std::endl;
iorFile.close();
std::cout << "IOR written to file Messenger.ior" << std::endl;

// Accept requests from clients.
orb->run();
orb->destroy();
}
catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
    return 1;
}

return 0;
10.2.6 Running the Application

Run the MessengerServer and MessengerClient as before. The MessengerServer will write a stringified object reference to the file named Messenger.ior. The MessengerClient will read the stringified object reference from the file, convert it to an object reference, and use it to invoke operations upon the Messenger object.

10.2.7 Program Output

Each operation invoked by the client will go through the client and server request interceptors that have been registered with their respective ORBs. The client’s output should look something like this:

```plaintext
> ./MessengerClient
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
Calling send_request().
User’s Name: Ron Klein
Calling receive_reply().
```

The server’s output should look something like this:

```plaintext
> ./MessengerServer
Calling ServerInterceptor constructor.
Calling ServerInterceptor name() method
IOR written to file Messenger.ior
Calling receive_request_service_contexts().
Calling receive_request().
Permission Granted
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
Calling send_reply().
```

10.3 Marshaling and the Service Context

Recall that data in the service context is a sequence of octets or raw bytes. In the previous example, where it was intended that the client and server run on the same machine, no special care was taken to ensure the integrity of the data in the service context. Due to different hardware having different endianess, the data in the service context can be corrupted if the client and server exist on
different machines. To keep this from occurring, octet sequences are converted to a network byte order, or *marshaled*. This is accomplished by the Codec.

### 10.3.1 The Codec

The Codec interface encodes and decodes between a sequence of octets and a `CORBA::Any`. A codec is obtained from the CodecFactory. There are multiple ways to obtain a reference to it. If the ORB is available, a reference to the CodecFactory can be obtained from a call to `ORB::resolve_initial_references("CodecFactory")`. During interceptor initialization, when the ORB is not available, a reference to the CodecFactory can be obtained from `ORBInitInfo::codec_factory()` or `ORBInitInfo::resolve_initial_references("CodecFactory")`.

The Codec has four distinct functions that encode/decode either the value of the data or the value and typecode of the data.

```cpp
local interface Codec
{
    exception InvalidTypeForEncoding {};
    exception FormatMismatch {};
    exception TypeMismatch {};

    CORBA::OctetSeq encode (in any data) raises (InvalidTypeForEncoding);
    any decode (in CORBA::OctetSeq data) raises (FormatMismatch);
    CORBA::OctetSeq encode_value (in any data) raises (InvalidTypeForEncoding);
    any decode_value (in CORBA::OctetSeq data, in CORBA::TypeCode tc)
        raises (FormatMismatch, TypeMismatch);
};
```

### 10.3.2 Example: Using the Codec

Encoding data with a codec is straightforward. First, a reference to the CodecFactory is obtained. Then an encoding scheme is specified and used to create a codec instance. This codec object can encode and decode between a `CORBA::Any` and a `CORBA::OctetSeq`.

This example extends the first interceptor example from 10.2.5 by encoding all the request information. The simple authentication scheme has changed from user name to user and group ids (uid/gid), which are passed as `CORBA::Long`, and would be corrupted if client and server reside on different-endian hosts. This example only shows the codec-specific code, as
the majority of the source has not changed. The complete source code for this example is in the TAO source code distribution in the directory

### 10.3.2.1 The Client
The user id is encoded before the send_message() function is invoked. First we obtain an initial reference to the CodecFactory, then we obtain a codec by passing an encoding scheme to the CodecFactory reference.

```cpp
// Obtain a reference to the CodecFactory.
CORBA::Object_var obj2 = orb->resolve_initial_references ("CodecFactory");
if (CORBA::is_nil(obj2.in())) {
    std::cerr << "Error: codec_factory" << std::endl;
    return 1;
}
IOP::CodecFactory_var codec_factory = IOP::CodecFactory::_narrow (obj2.in());
std::cout << "got codec factory" << std::endl;

// Set up a structure that contains information necessary to
// create a GIOP 1.2 CDR encapsulation Codec.
IOP::Encoding encoding;
encoding.format = IOP::ENCODING_CDR_ENCAPS;
encoding.major_version = 1;
encoding.minor_version = 2;

// Obtain the CDR encapsulation Codec.
IOP::Codec_var codec = codec_factory->create_codec (encoding);

The user id is inserted into a CORBA::Any which is encoded and returns a
CORBA::OctetSeq that is used in the call to send_message().

```cpp
// our user id
CORBA::Long uid = 64321;
CORBA::Any uid_as_any;
uid_as_any <<= uid;
CORBA::OctetSeq client_uid = *codec->encode (uid_as_any);
messenger->send_message (client_uid);
```

### 10.3.2.2 The Client Interceptor
In this example, the client interceptor passes group id information. Since the gid is also a CORBA::Long, it too is marshaled. Recall that interceptors are
registered within the call to ORB_init(). Since an ORB reference is not yet available, interceptors that need access to a codec must obtain it from operations in the ORBInitInfo interface.

```cpp
void
ClientInitializer::post_init (
    PortableInterceptor::ORBInitInfo_ptr info)
throw (CORBA::SystemException)
{
    // get Codec factory
    IOP::CodecFactory_var codec_factory = info->codec_factory();
}
```

The interceptor gets the CodecFactory reference through an argument passed to its constructor by the ORBInitializer.

```cpp
// Create and register the request interceptors.
PortableInterceptor::ClientRequestInterceptor_var ci =
    new ClientInterceptor (codec_factory);
info->add_client_request_interceptor (ci.in());
```

The client interceptor constructor has been modified to accept the reference to the CodecFactory and create a codec. The send_request() interception point will use the codec to encode the gid. It then adds the encoded octet sequence to the service context. The marshaling code is identical to that in the client.

```cpp
ClientInterceptor::ClientInterceptor (IOP::CodecFactory_var cf)
    : myname_ ("Client_Authentication_Interceptor")
{
    std::cout << "Calling ClientInterceptor constructor." << std::endl;

    // Set up a structure that contains information necessary to
    // create a GIOF 1.2 CDR encapsulation Codec.
    IOP::Encoding encoding;
    encoding.format = IOP::ENCODING_CDR_ENCAPS;
    encoding.major_version = 1;
    encoding.minor_version = 2;

    // Obtain the CDR encapsulation Codec.
    this->codec = cf->create_codec (encoding);
}
```

```cpp
void
ClientInterceptor::send_request (
    PortableInterceptor::ClientRequestInfo_ptr ri)
```
throw (CORBA::SystemException, PortableInterceptor::ForwardRequest)
{
    std::cout << "Calling send_request()." << std::endl;

    IOP::ServiceContext sc;
    sc.context_id = service_ctx_id;

const CORBA::Long gid = 9007;
    std::cout << "GID: " << gid << std::endl;

    CORBA::Any gid_as_any;
    gid_as_any <<= gid;

    sc.context_data = ACE_reinterpret_cast(
        IOP::ServiceContext::_tao_seq_Octet &,
        *codec->encode(gid_as_any));

    // Add this context to the service context list.
    ri->add_request_service_context (sc, 0);
}

10.3.2.3 The Server

The octet sequence is decoded in the send_message() function of the server. The code is very similar to the code in the client. A reference to the CodecFactory is passed an encoding structure and returns a codec that is used to demarshal the uid. In practice, creating the codec could be moved to the constructor, but in this example it is left in the send_message() function for clarity.

    // Obtain a reference to the CodecFactory.
CORBA::Object_var obj = orb->resolve_initial_references ("CodecFactory");
    if (CORBA::is_nil(obj.in()))
    {
        std::cerr << "Error: codec_factory" << std::endl;
        return 1;
    }

    IOP::CodecFactory_var codec_factory = IOP::CodecFactory::_narrow (obj.in ());
    std::cout << "Server got codec factory" << std::endl;

    // Set up a structure that contains information necessary to
    // create a GIOP 1.2 CDR encapsulation Codec.
    IOP::Encoding encoding;
    encoding.format = IOP::ENCODING_CDR_ENCAPS;
    encoding.major_version = 1;
    encoding.minor_version = 2;
// Obtain the CDR encapsulation Codec.
IOP::Codec_var codec = codec_factory->create_codec (encoding);

CORBA::Any uid_as_any;
uid_as_any = *(codec->decode(user_name));

CORBA::Long uid;
uid_as_any >>= uid;
std::cout << "UID: " << uid << std::endl;

10.3.2.4 The Server Interceptor

As in the client, the server interceptor is not able to access the CodecFactory through the ORB because an ORB reference is not available. Instead, a reference to the CodecFactory is obtained in the ORBInitializer from ORBInitInfo::codec_factory(). This reference is passed as an argument to the interceptor’s constructor. The code that does this is identical to the client code and is not shown here.

Decoding in the server interceptor is more complex than the decoding in the server’s send_message() function. Previously we decoded an octet sequence that was passed as a parameter to send_message(). At the receive_request() interception point, we do not have direct access to an octet sequence and must extract it from the service context. There is no service context member function that returns the resident data as an octet sequence, but there is a way to construct an octet sequence from the data in the service context.

// need to construct an octet seq for decoding
CORBA::OctetSeq ocSeq = CORBA::OctetSeq(
    sc->context_data.length(),
    sc->context_data.length(),
    sc->context_data.get_buffer(),
    0);

Once the data has been obtained as a CORBA::OctetSeq, it can be decoded using the codec that was created in the interceptor’s constructor.

CORBA::Any gid_as_any;
gid_as_any = *codec->decode(ocSeq);

CORBA::Long gid;
gid_as_any >>= gid;
for (int i=0; i<3; ++i) {
    if (gid == allowed_gid[i]) {
        permission_granted = true;
    }
}

10.3.3 Program Output
The output of this example is similar to the previous example. The client output should resemble:

> ./MessengerClient
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
got codec factory
Calling send_request().
GID: 9007
Calling receive_reply().
message was sent

The server output should resemble:

> ./MessengerServer
Calling ServerInterceptor constructor.
Calling ServerInterceptor name() method
IOR written to file Messenger.ior
Calling receive_request_service_contexts().
Calling receive_request().
Permission Granted
Server got codec factory
UID: 64321
Calling send_reply().

10.4 IOR Interceptors
A second class of interceptors, IOR Interceptors, can add user-defined information (called tagged components) to an object’s IOR at the time of the creation of the IOR. This is useful in cases where developers might want to add information about a server’s or object’s capabilities or requirements. IOR interceptors implement the following interfaces, which are defined in $TAO_ROOT/tao/PortableInterceptor.pidl.

local interface IORInterceptor : Interceptor
The server calls `establish_components()` for all registered IOR interceptors in the course of assembling the data that will be included in an IOR, adhering to the CORBA specification, ignores any exceptions thrown by `establish_components()`. The `IORInfo` interface is used to get profile information and to add components to an IOR.

### 10.4.1 Defining and Implementing the IOR Interceptor

An IOR interceptor multiply inherits from `PortableInterceptor::IORInterceptor` and `TAO_Local_RefCounted_Object`. It must implement the `establish_components()`, `name()`, and `destroy()` methods.

```cpp
#include "tao/PortableInterceptorC.h"
#include "tao/LocalObject.h"

class ServerIORInterceptor : 
    public virtual PortableInterceptor::IORInterceptor, 
    public virtual TAO_Local_RefCounted_Object 
{

    public:
    virtual char* name () throw (CORBA::SystemException);

    virtual void destroy () throw (CORBA::SystemException);

    virtual void establish_components (PortableInterceptor::IORInfo_ptr info) throw (CORBA::SystemException);

};
```
10.4.2 Registering the IOR Interceptor
Similar to request interceptors, IOR Interceptors are registered in the `pre_init()` or `post_init()` methods of the ORBInitializer. A newly created instance of `IORInterceptor` is registered by passing it as an in parameter to `ORBInitInfo::add_ior_interceptor()`.

10.4.3 Extracting Tagged Information
Tagged components in an IOR can be extracted on the client-side at any of the four implemented client interception points (`send_poll()` is not supported). The tagged component of an IOR is returned from `ClientRequestInfo::get_effective_component()`.

10.4.4 Example: “ServerRequiresAuth” Tag in IOR
The previous examples in 10.2.5 and 10.3.2 show how interceptors can be used to provide simple authentication. In a more complex environment, different levels of security would exist and clients would not necessarily know what types of authentication they might need to provide with remote requests.

This example demonstrates how IOR interceptors can allow an object to advertise the type of authentication it requires. The following code extends the `codec` example from 10.3.2 on the server side by publishing the string "ServerRequiresAuth" as a component tag in the IOR. The client extracts this new tag. The complete source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/PortableInterceptors/IOR`.

10.4.4.1 Developing the IOR Interceptor
The `name()` and `destroy()` methods of the `IORInterceptor` are simple.

```c++
char* ServerIORInterceptor::name ()
    throw (CORBA::SystemException)
{
    return CORBA::string_dup ("ServerIORInterceptor");
}

void ServerIORInterceptor::destroy ()
    throw (CORBA::SystemException)
{
}
```
To implement `establish_components()`, we create an `IOP::TaggedComponent` and choose an arbitrary numeric value for its `tagID`.

**Note**  
Were we developing an application that needed to interface with other CORBA applications, we would want to make sure our `tagID` was unique, and would ask the OMG to assign a Component ID. See ftp://ftp.omg.org/pub/docs/ptc/99-02-01.txt for additional information.

We then copy the string “ServerRequiresAuth” into the `TaggedComponent` buffer and call `add_ior_component()`.

```cpp
void ServerIORInterceptor::establish_components(
    PortableInterceptor::IORInfo_ptr info )
throw (CORBA::SystemException)
{
    const char * permission = "ServerRequiresAuth";

    // arbitrary tag.
    CORBA::ULong tagID = 9654;

    // populate the tagged component
    IOP::TaggedComponent myTag;
    myTag.tag = tagID;
    myTag.component_data.length (ACE_OS::strlen(permission) + 1);
    CORBA::Octet *buf = myTag.component_data.get_buffer();
    ACE_OS::memcpy (buf, permission, ACE_OS::strlen(permission) + 1);

    // add the tagged component
    info->add_ior_component (myTag);
    std::cout << "Created Tagged IOR." << std::endl;
}
```

### 10.4.4.2 Installing the IOR Interceptor

To install the IOR interceptor, we implement the `post_init()` method of the `ServerInitializer` class. This method is similar to the `post_init()` method previously shown for Request interceptors. The IOR interceptor is registered by creating a new instance of the interceptor and passing it to the `add_ior_interceptor()` method.

```cpp
void ServerInitializer::post_init (  
```
PortableInterceptor::ORBInitInfo_ptr info)
throw (CORBA::SystemException)
{
    // get reference to the codec_factory
    IOP::CodecFactory_var codec_factory = info->codec_factory();

    // Create and register the request interceptors.
    PortableInterceptor::ServerRequestInterceptor_var si =
        new ServerInterceptor(codec_factory);
    info->add_server_request_interceptor (si.in());

    // add IOR Interceptor
    PortableInterceptor::IORInterceptor_var iori = new ServerIORInterceptor;
    info->add_ior_interceptor (ior.in());
}

10.4.4.3 Decoding the Tag in the Client
In this example, the client accesses the IOP::TaggedComponent information at the send_request() interception point. It retrieves an IOP::TaggedComponent based upon a known tagID. If the TaggedComponent does not exist, the operation get_effective_component() raises a CORBA::BAD_PARAM exception, hence the try/catch block below.

    const CORBA::ULong tagID = 9654;
    try {
        IOP::TaggedComponent_var myTag = ri->get_effective_component(tagID);
        char *tag =
            ACE_reinterpret_cast (char *, myTag->component_data.get_buffer());
        std::cout << "IOR Tag is : " << tag << std::endl;
    } catch(CORBA::BAD_PARAM&) {
        std::cerr << "Tagged Component not found" << std::endl;
    }

10.4.5 Program Output
The server generates a tagged IOR which can be viewed in a more readable form using the utility catior (in $TAO_ROOT/utils/catior):
>
$TAO_ROOT/utils/catior/catior -f Messenger.ior
reading the file Messenger.ior

here is the IOR
IOR:010000001200000049444c3a4d657373656e6765723a312e3000000000100000000000000000870
00000101020010000000636869702e6f63697765622e636f6d00878200001b0000014010f0052
decoding an IOR:
The Byte Order: Little Endian
The Type Id: "IDL:Messenger:1.0"
Profile Count: 1
IIOP Version: 1.2
   Host Name: chip.ociweb.com
   Port Number: 33415
   Object Key len: 27
   Object Key as hex:
   14 01 0f 00 52 53 54 0e 02 57 3c 69 8d 0c 00 00
   00 00 00 01 00 00 00 01 00 00 00
   The Object Key as string:
....RST..W<i................
   The component <0> has tag <0>
      Component Value len: 8
      Component Value as hex:
      01 00 00 00 4f 41 54
      The Component Value as string:
.....OAT
   The component <1> has tag <1>
      Component Value len: 20
      Component Value as hex:
      01 35 b3 40 01 00 01 00 00 00 00 00 09 01 01 00
      00 00 00 00
      The Component Value as string:
..5................
   The component <2> has tag <9654>
      Component Value len: 19
      Component Value as hex:
      53 65 72 76 65 72 52 65 71 75 69 72 65 73 41 75
      74 68 00
      The Component Value as string:
ServerRequiresAuth.

At the send_request() interception point, the client obtains the TaggedComponent and casts its to a char* for output. The output of the client should resemble:

./MessengerClient
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
got codec factory
Calling send_request().
IOR Tag is : ServerRequiresAuth
GID: 9007
10.5 The PortableInterceptor::Current

The PortableInterceptor::Current or PICurrent is a slot table that is used to transfer thread context information between the request and reply service contexts. The PICurrent is an ancillary object to Portable Interceptors. Its use is not required, but it is helpful in propagating data when the service context is not available or not yet available.

10.5.1 Using PICurrent

The PortableInterceptor::Current interface is defined in $TAO_ROOT/tao/PortableInterceptor.pidl:

```plaintext
module PortableInterceptor {
    typedef unsigned long SlotId;
    exception InvalidSlot {}
    local interface Current : CORBA::Current {
        any get_slot (in SlotId id) raises (InvalidSlot);
        void set_slot (in SlotId id, in any data) raises (InvalidSlot);
    }
};
```

A reference to the PICurrent is obtained from a call to ORB::resolve_initial_references("PICurrent"). Data in the form of a CORBA::Any is inserted into a slot with set_slot() and is retrieved with get_slot().

10.5.2 When to use PICurrent

There are special instances where PICurrent can be helpful on both the server- and client-side. Recall that the receive_request_service_context() interception point does not have access to the service context. Any data that is needed at this interception point can be copied into the PICurrent before the invocation and will be available at the interception point. The CORBA specification discusses a client-side case where the PICurrent is useful in stopping recursion. The following
example demonstrates how interceptors can be called recursively and how the PiCurrent can be used to pass a flag between the client and server service contexts to allow the client to recognize that it is making a recursive call.

10.5.3 **Example: Stopping Client-side Recursion**

Infinite recursion can happen if an interceptor makes an ORB-mediated invocation on a CORBA object. Suppose the client makes an invocation, which calls `send_message()`, and `send_message()` makes its own invocation, which will call `send_message()` indefinitely until somehow the interceptor realizes it is recursing or the application crashes. The PiCurrent can be used to pass a flag that the ClientRequestInterceptor can use to keep from making recursive calls.

In this example, we assume that the client needs to know the server’s date and time with each invocation operation. Rather than returning the date and time from a call to `send_message()`, we choose to extend our IDL to add a `get_time()` function. The complete source code for this example is in the TAO source code distribution in the directory $TAO_ROOT/DevGuideExamples/PortableInterceptors/PICurrent.

Here is our Messenger interface with the addition of a `get_time()` operation:

```idl
interface Messenger
{
    boolean send_message ( in  string user_name,
                           in  string subject,
                           inout string message);

    string get_time ();
};
```

The `get_time()` operation is implemented on the server as:

```c
char* Messenger_i::get_time ()
throw (CORBA::SystemException)
{
    time_t thetime;
    struct tm* timeinfo;

    ACE_OS::time(&thetime);
    timeinfo = ACE_OS::localtime(&thetime);
    char* timestring = CORBA::string_dup(ACE_OS::asctime(timeinfo));

    return timestring;
}
```
The functionality we would like to see in the client is that every time
Messenger::send_message() is called, Messenger::get_time() is
called. If we add a call to get_time() in the send_request() interception
point, we will cause send_request() to be called recursively. Using the
PICurrent, we can detect this recursion.

Before we can use the PICurrent, we must get a reference to it. This is done
in the ORBInitializer. Because we want to call get_time(), we need a
reference to the Messenger object. In the earlier examples in this chapter, we
read an IOR from a file and call ORB::string_to_object() to obtain the
object reference, but interceptors do not have access to the ORB in the
ORBInitializer, so string_to_object() is not available. For this
example we made the Messenger object an initial reference by passing the
server’s IOR as part of the -ORBInitRef command line argument.
Alternately, we could bind the Messenger object in the Naming Service and
resolve the Naming Service as an initial reference in the ORBInitializer.

```cpp
void ClientInitializer::post_init (
    PortableInterceptor::ORBInitInfo_ptr info)
throw (CORBA::SystemException)
{
    // resolve Messenger object
    CORBA::Object_var obj = info->resolve_initial_references ("Messenger");
    Messenger_var messenger = Messenger::_narrow (obj.in());
    if (CORBA::is_nil(messenger.in())) {
        std::cerr << "Not a Messenger reference" << std::endl;
        // We could throw an exception here, or just ignore the error and go on.
    }

    // allocate slot in the current for our use. Since we cannot obtain a
    // reference to the PICurrent through
    // ORB::resolve_initial_references(), we obtain it through
    ORBInitInfo::resolve_initial_references().

    // allocate slot
    PortableInterceptor::SlotId slot = info->allocate_slot_id();

    // get PICurrent
    CORBA::Object_var current_obj = info->resolve_initial_references ("PICurrent");
    PortableInterceptor::Current_var pic =
        PortableInterceptor::Current::_narrow (current_obj.in()));
```
A CORBA::Boolean serves as our recursion flag. Initially there is no recursion, so we set a false value in the PICurrent slot and finish installing the interceptor.

```
// set recursion flag
CORBA::Any flag;
CORBA::Boolean x = 0;
flag <<= CORBA::Any::from_boolean(x);

pic->set_slot(slot, flag);
```

In the client interceptor, the code pertinent to Messenger::send_message() is unchanged.

```
void ClientInterceptor::send_request (PortableInterceptor::ClientRequestInfo_ptr ri)
throw (CORBA::SystemException, PortableInterceptor::ForwardRequest)
{
  std::cout << "Calling send_request()." << std::endl;

  IOP::ServiceContext sc;
  sc.context_id = service_ctx_id;

  const char user_name[] = "Ron Klein";
  std::cout << "User's Name: " << user_name << std::endl;
  CORBA::ULong string_len = sizeof (user_name) + 1;
  CORBA::Octet *buf = 0;
  ACE_NEW (buf, CORBA::Octet[string_len]);
  ACE_OS::strcpy (ACE_reinterpret_cast(char *, buf), user_name);

  sc.context_data.replace (string_len, string_len, buf, 1);

  Before calling get_time(), the interceptor retrieves the flag from the PICurrent. If it is false, the interceptor inserts a value of true, which will stop the recursion on the next invocation. Then it calls get_time(). Finally, before exiting send_request() the value in the PICurrent is reset to false, so that later invocations will call get_time().

  // recursive call setup
  CORBA::Any* recurse = ri->get_slot(slot);
```
CORBA::Boolean x;
*recurse >>= CORBA::Any::to_boolean(x);

CORBA::Any flag;
if (x == 0) {
    x = 1;
    flag <<= CORBA::Any::from_boolean(x);

    pic->set_slot(slot, flag);

    // get server time
    std::cout << "Server Time = " << messenger->get_time() << std::endl;
}

// Add this context to the service context list.
ri->add_request_service_context (sc, 0);

// reset recursion test
x = 0;
flag <<= CORBA::Any::from_boolean(x);
pic->set_slot(slot, flag);

10.5.4 Program Output
The server output should resemble:

> ./MessengerServer
Calling ServerInterceptor constructor.
Calling ServerInterceptor name() method

IOR written to file Messenger.ior
Calling receive_request_service_contexts().
Calling receive_request().
Permission Granted
Calling send_reply().
Calling receive_request_service_contexts().
Calling receive_request().
Permission Granted
Message from: TAO User
Subject: TAO Test
Message: Hello!
Calling send_reply().

The client output should resemble:

> ./MessengerClient -ORBInitRef Messenger=file://Messenger.ior
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
Calling send_request().
User’s Name: Ron Klein
Calling send_request().
User’s Name: Ron Klein
Calling receive_reply().
Server Time = Tue Jan 29 13:19:16 2002

Notice that send_request() is called twice. The first is from the send_message() operation invocation. The second is from the get_time() operation invocation.

10.6 Summary

This chapter showed how to develop applications using TAO and its CORBA 3.0.3 compliant Portable Interceptors implementation. Several aspects of Portable Interceptors were discussed, including:

- Portable Interceptors are used to monitor and modify transparently the requests and replies between clients and server. They can be implemented with few modifications to existing code.
- Codecs (coders/decoders) can be used to marshal the service context, so that byte order differences between systems do not corrupt request/reply data.
- IOR Interceptors are used to add tagged components to IORs. This allows servers and other objects to advertise their capabilities or requirements to clients.
- The PICurrent is a table that can be used to transfer data between the reply and request service contexts.
CHAPTER 11

Value Types

11.1 Introduction

The original CORBA specification focused on the challenges of remotely
invoking operations on objects, regardless of location. A number of data types
were introduced to allow rich interfaces to be defined, but a strict separation
was maintained between the functionality of a system, specified via interfaces,
and the data of a system, specified via data types. With value types (see
Chapter 5 the CORBA Core specification, OMG Document formal/04-03-12),
it is possible to define objects in IDL that contain both functionality and data
(called state members) that can be passed remotely by their value. These value
types are defined using the valuetype keyword. Value types can be helpful
for the case where an object role is primarily to hold data, or it is common to
copy an object with a large amount of data.

This chapter provides an introduction to the subset of value type features that
are implemented in TAO. For additional information about value types, see
Chapter 11 of Pure CORBA.
11.2 Uses for Value Types

A value type can be used in several situations that may otherwise be awkward in CORBA. You can use a value type as a replacement for an IDL struct, as another type of local interface, or as a way to combine data with operations, like a C++ class. They can also be passed through any of the TAO CORBA event services such as the Notification Service described in Chapter 27.

Using a value type in place of a struct allows you to use inheritance as a way to organize your structures. This is easier to work with than building complex structures using containment, because it can be tedious to access many levels of nested structs.

In many cases, using value types can be more straightforward and more efficient using than CORBA local interfaces. An operation invocation on a value type does not go through an ORB. Public accessor and mutator operations on value types do not transfer ownership, so there is no need for _var types and reference counting, but operations and attributes still have typical interfaces. Unlike a local interface, a value type is not derived from CORBA::Object, and therefore avoids considerable compilation and memory overhead.

Note

*It is important to keep in mind that for each language used in your distributed system an implementation of your value type must be made. Each language implementation must behave identically for your value type for work correctly.*

Perhaps the most common use of a value type is for passing objects by value. A normal CORBA interface leaves the data portion of an object undefined, allowing you to implement the interface operations with whatever servant best suits the needs of your application. By using value types, your IDL interface can constrain the data types, thus trading a little flexibility in how the servant is implemented for having the ORB pass your objects by value. With value types, you are still required to provide an implementation for the interface.

Although TAO does not support the eventtype keyword, you can pass value types through any of the included event services, or through any custom interface that passes values using Any. Your value type implementation need
only be linked into the code that ultimately extracts the value type from the Any.

Note  
*The CORBA eventtype is a specialization of the valuetype.*

When a value type is passed from one process to another, only the state of the object is passed. It is the responsibility of the receiving process to provide an implementation of that value type, as well as a factory suitable for reconstructing the object from the data marshalled using CORBA. A base factory class will be generated and will contain virtual methods corresponding to each of the factory operations defined in the value type IDL. Processes operating on a value type are not required to use the same implementation. You must register a factory object with the ORB using `CORBA::ORB::register_value_factory()`.

Note  
*Value types with only state members will have a factory generated automatically, although you must still register an instance of this factory with the ORB.*

Value types break one of CORBA’s original tenets, that the IDL is all that is required for different processes to communicate, regardless of implementation language, architecture, or location. When using value types, you are required to supply value type implementations, which may restrict the processes and implementations with which you can communicate. Although this restriction is often offset by the potential efficiency of using value types, it sometimes forces developers to avoid the use of value types in their applications.

### 11.3 Defining Value Types in IDL

A value type in IDL can be thought of as a hybrid of a struct and an interface. Like a struct, the value type can have data members, and passes all of its enclosed data across an interface. Like an interface, a value type defines a set of operations that can be called upon objects of that type. In addition, value types support inheritance allowing for the definition of complex models.
Value Types

Note Currently TAO does not support inheritance from CORBA interfaces using the supports keyword.

For example, here is a definition for a simple value type:

```plaintext
valuetype Person {
    public string name;
    public long balance;

    factory create(in string name);

    void debit(in long amt);
};
```

The state members of a value type require the private or public keyword to indicate the accessibility of the generated accessors and mutators. The factory keyword is used to denote a special operation that will be used to construct a Person. A value type can contain more than one factory declaration, and the factory can have any name.

Note It is often unnecessary to declare factory methods in IDL, because you can simply use normal C++ constructors to create the value type instances.

The value type can be used with an IDL operation:

```plaintext
interface Transaction {
    void update(in Person p);
};
```

As the Person object is passed via the update() operation, the entire object (including its state) is copied from the client to the server. This means that a Person object exists in both the client and server, and invocations in a particular process only affect the state of the object local to that process.

11.4 A Value Type Example

In this section we show an example of using value types that is based on the Messenger example introduced in Chapter 3.
11.4 A Value Type Example

11.4.1 Implementing Value Types

The first step in implementing value types is to use the IDL compiler to generate the usual skeleton and stub C++ files from an IDL file defining one or more value types and interfaces that use them. The IDL compiler generates all the code required for value type support. It will generate the C++ classes Message and OBV_Message (OBV stands for object-by-value) as well as POA_Messenger. The IDL compiler also generates Message_var, Message_out, and Message_ptr types like those generated for interfaces. The complete source code for this example can be found in the source code distribution in $TAO_ROOT/DevGuideExamples/ValueTypes/Messenger.

11.4.1.1 The valuetype IDL

The IDL for our example defines a simple Message value type which we pass through a modified version of our Messenger interface. We provide attributes and operations that will presumably give access to the internal data. This example is meant to illustrate available features more than correct design. We also forgo specification of a factory interface, because we create our factory from the default base class.

valuetype Message {
    typedef sequence<string> AddrList;

    private AddrList addrs_;    
    private string user_;       
    private string subject_;    
    private string text_;        

    AddrList getAddresses();    
    void addAddress (in string address); 
    attribute string user;      
    attribute string subject;   
    attribute string text;      
    void print ();               
};

interface Messenger {
    boolean send (inout Message msg);
};
Implementing the Value Type Class

Implementing a value type class is analogous to implementing a servant class for an interface. The IDL compiler generates an `OBV_Message` class from which our implementation class must inherit. The `OBV_Message` class itself inherits from the generated `Message` class and includes the state member accessor/mutator functions. Think of the `OBV_Message` class as a partial implementation of the abstract `Message` class.

Note

The private and public qualifiers for state members to not directly map to the C++ private and public keywords. For all state members the IDL compiler generates pure virtual accessor and modifier functions. For public members these functions map to public C++ functions. For private members these functions map to protected C++ functions.

It is up to us to provide implementations for the value type state members and operations. In our example, we need to provide implementations for the user, subject, and text attributes, as well as the getAddresses() and addAddress() operations.

Additionally, all value type objects use intrusive reference counting. The `ValueBase` class, from which all value types derive, defines pure virtual `_add_ref()`, `_remove_ref()`, and `_refcount_value()` functions. The easiest way to provide a correct implementation for these is to mix in the `CORBA::DefaultValueRefCountBase` type. This is analogous to mixing in `RefCountServantBase` for servant classes.

Here is a portion of the code for our `Message` implementation class:

```c++
// Message_i.h
class MessageImpl : public virtual OBV_Message
                    , public virtual CORBA::DefaultValueRefCountBase
{
  public:
    MessageImpl (const char* address, const char* user, const char* subject,
                const char* txt);

    virtual char* user();
    virtual void user(const char*);
    // ...
};
```
// Message_i.cpp
MessageImpl::MessageImpl (const char* address, const char* user,
                          const char* subject, const char* txt)
{
    addAddress(address);
    user_(user);
    subject_(subject);
    text_(txt);
}

char* MessageImpl::user()
{
    return CORBA::string_dup(user_());
}

void MessageImpl::user(const char* s)
{
    user_(s);
}

Notice that, in our constructor, we use the modifier functions generated in the
OBV_Message class because there is no OBV_Message constructor to which
we can delegate.

---

**Note**  
This is a flaw in the current TAO implementation, as the specification for
value types requires a generated constructor that accepts a parameter for
each state member of the value type in the order in which they are declared in
IDL.

---

### 11.4.1.3 Implementing the Value Type Factory

A value type factory is required to allow the ORB to create and demarshal
value type objects that are included in GIOP Request and Reply messages.
This factory knows how to create value type objects of a specific type that are
registered with the ORB.

We must implement a factory class derived from the ValueFactoryBase
defined in TAO. If we had not specified any operations or attributes in our
value type IDL, then a complete factory class called Message_init would
have been generated automatically. If we had defined any factories in the IDL,
then a base class called Message_init would have been generated, and we
would have to provide implementation for each of its virtual methods. Here is
the definition for the factory:
Value Types

#include <tao/Valuetype/ValueFactory.h>

// Message_i.h
class MessageFactory : public virtual CORBA::ValueFactoryBase
{
public:
    static void register_new_factory(CORBA::ORB& orb);
    virtual CORBA::ValueBase* create_for_unmarshal();
};

We add the static register_new_factory() function to make it easier to register the factory with the ORB. We call this function from both the server and client main() functions. The MessageFactory definition is as follows:

// Message_i.cpp
void MessageFactory::register_new_factory (ORB& orb)
{
    CORBA::ValueFactoryBase_var mf = new MessageFactory;
    CORBA::String_var id = ::_tc_Message->id();
    orb.register_value_factory(id.in(), mf.in());
}

CORBA::ValueBase* MessageFactory::create_for_unmarshal ()
{
    return new MessageImpl;
}

The register_new_factory() operation simply associates the Repository ID of our factory class with an instance of our factory class. Later, when a Message object is received, the ORB will use the create_for_unmarshal() operation to create a local implementation for the Message object.

A value type factory only needs to be registered with orbs that need to unmarshal value types. Often this means that you can forgo this registration on the client side, although in our example the Message object is passed through an inout parameter, and therefore both sides must register a factory.

11.4.2 Using Value Types

A value type is typically referenced through a _var smart pointer that manages memory allocated for a value type in a manner similar to the _var smart pointer types used for CORBA object proxies. You can call the operations on a value type by simply calling the corresponding member
function on the value type object. Public state members are accessed and modified using member functions with the same name as the state member. In our example, the client creates a `Message`, then sends it to the server. The server simply calls the `Message::print()` function, then uses the same `Message` object to reply to the client.

Here is our implementation class for the `Messenger` interface:

```cpp
// Messenger_i.h
class Messenger_i : public virtual POA_Messenger
{
 public:

 virtual CORBA::Boolean send_message (Message*& msg)
     ACE_THROW_SPEC ((CORBA::SystemException));
};

// Messenger_i.cpp
CORBA::Boolean Messenger_i::send_message (Message*& msg)
    ACE_THROW_SPEC ((SystemException))
{
    // print the message sent from the client
    msg->print();

    // populate the return message
    msg->user("Son");
    msg->addAddress("Mom");
    msg->addAddress("Dad");
    std::ostringstream out;
    CORBA::String_var subject = msg->subject();
    out << "RE: " << subject.in();
    msg->subject(out.str().c_str());
    msg->text("Ok. I'm on my way.");

    return 1;
}
```

### 11.5 An Example using Value Types as Events

Although the `eventtype` keyword is not supported by TAO, it is essentially just a synonym for `valuetype`. This means that it can be convenient to pass value types as events using the TAO Event Service, RT Event Service, or Notification Service.
This capability was not working in early releases of 1.4a, but was fixed for patch release 5.

The complete source code for this example can be found in the source code distribution in $TAO_ROOT/DevGuideExamples/ValueTypes/Notify.

### 11.5.1 Event Example IDL

```idl
valuetype MyEvent
{
    public string name;
    public long kind;
    typedef sequence<long> LongSeq;
    private LongSeq payload;

    void dump();
    long size();
    void add_long(in long n);
};
```

### 11.5.2 Event Example Supplier

The following is an excerpt from an example showing how to use a value type with the Notification Service CosEvent interfaces. The supplier encapsulates the MyEvent value type in an Any before pushing it to the Notification Service channel using the PushConsumer interface.

```java
... MyEvent_var event_ = new MyEvent_i("TestName", 42);
...
bool push_next_event() {
    try {
        if (! connected_) {
            ACE_DEBUG((LM_DEBUG, "Trying to push when disconnected.\n"));
            return false;
        }
        ACE_DEBUG((LM_DEBUG, "+");

        ++event_count_;

        Any a;
        a <<= event_;
        consumer_->push(a);
    }
```
if (event_count_ >= num_events_ && num_events_ > 0) {
    ACE_DEBUG((LM_DEBUG, "Stopping after %d events.\n", event_count_));
    disconnect();
} else {
    schedule_next_event(EVENT_DELAY);
}
return true;
} catch (CORBA::Exception& e) {
    ACE_PRINT_EXCEPTION(e, "TestSupplier::push_next_event()"uellement au client.
}
return false;

11.5.3 Event Example Consumer
The following is an excerpt from the consumer code, showing how the value

type is extracted from the interface. The consumer code must register a value
factory with the ORB, or the extraction will trigger a CORBA::UNKNOWN
system exception.

... virtual void push(const Any& a)
    throw (SystemException, CosEventComm::Disconnected)
{
    MyEvent* vt;
    a >>= vt;

    ACE_DEBUG((LM_DEBUG,
        "\nReceived MyEvent name=%s, kind=%d, size=%d\n",
        vt->name(), vt->kind(), vt->size()));
    vt->dump();

    if ( ++ event_count_ >= num_events_ && num_events_ > 0) {
        ACE_DEBUG((LM_DEBUG, "\nDisconnect after %d events.\n", event_count_));
        disconnect();
    }
}

CORBA::ValueFactoryBase_var factory = new MyEventFactory;
String_var id = _tc_MyEvent->id();
orb->register_value_factory(id.in(), factory.in());
11.6   Value Types and Inheritance

TAO currently supports value type inheritance from other value types only. It supports multiple inheritance from abstract value types (discussed later in this section), and single inheritance from regular non-abstract value types.

Note  TAO IDL also supports the definition of an abstract interface. An abstract interface can be used for defining base classes that can be the base class for interfaces or value types. However, this feature is not useful without the supports keyword. As mentioned earlier, this keyword is not currently allowed by the IDL compiler.

11.6.1   Regular Value Types

You can use inheritance between regular value types as you might have done with a struct in the past, and it can be used to support a common interface as with C++ inheritance. For example, in our application, there might be other value types that we wish to print:

```plaintext
valuetype Printable
{
    void print();
};

valuetype Message : Printable
{
    // ...
};

valuetype Document : Printable
{
    // ...
};
```

By moving the `print()` operation from our `Message` type to a `Printable` base type, we are able to add a new type that can print. This will allow `Message` and `Document` to be passed to any IDL operation that takes a `Printable` type. However, you must still implement the `print()` operation for `Message` and `Document`, because value type inheritance does not provide implementation inheritance.
Of course, we can also have base types that contain data. Here is an example of several value types that would have been extremely tedious to access with the CORBA struct mapping:

```java
valuetype Party
{
  public string name;
  public string address;
};

valuetype Person : Party
{
  public string birth_date;
};

valuetype Employee : Person
{
  public string date_of_hire;
};

valuetype Manager : Employee
{
  typedef sequence<Employee> Reports;
  public Reports reports;
};
```

If we had implemented these as nested structs, we would have had code that called such things as `mgr.employee.person.party.name = "Stan"`. Instead, we simply use `mgr.name()`.

### 11.6.2 Abstract Value Types

A value type may be defined as abstract. This allows the value type to be used as a base class in combination with another value type. An abstract value type may not contain any state members. However, it can contain operations and attributes. A value type may inherit from one value type, and many abstract value types. Using abstract value types, we could add another base type to our `Message` value type from the previous example:

```java
valuetype Named {
  public name;
};

abstract valuetype Printable {
  void print();
};
```
By adding the `abstract` keyword, we are able to derive `Message` from both `Named` and `Printable`.

### 11.7 TAO Compliance

TAO supports many of the basic features of value types, including:

- Inheriting one value type from another
- The use of abstract value types
- Most of the basic operations available to what *Pure CORBA* refers to as *regular value types*.

TAO’s support of value types does *not* include:

- Value boxes (also called boxed values).
- Custom marshaling.
- The `truncatable` keyword and truncatable inheritance.
- `ValueBase::_copy_value()`.
- Inheritance from IDL interfaces.
- `PortableServer::ValueRefCountBase`.
- `OBV_*` constructors.
- `ValueBase::_add_ref()` returning `ValueBase*`.
- `eventtype` keyword.

Work is continuing on OBV features in TAO, and some of the above features may be implemented within the near future. However, value types as they currently exist in TAO are a functional and valuable tool that you can take advantage of right now.
CHAPTER 12

Smart Proxies

12.1 Introduction

Smart proxies are a TAO-specific feature that allows customization of proxy behavior. Some other ORB implementations provide similar functionality, but there is no standard for smart proxies.

A smart proxy is an alternative class to the default proxy generated by the TAO IDL compiler. The purpose is to provide the client application developer the ability to extend the default behaviors. It is written by the client application developer and may contain member functions that never send requests to the target object.

A proxy supplies a client application with an interface to a target CORBA object that allows the client to access operations on the object in a location-transparent manner (as if the remote object resides in the client’s address space). When the TAO IDL compiler is invoked on an IDL interface, it generates a proxy (stub) class that has the same name as the interface. For each operation and attribute in the interface, the TAO IDL compiler generates a corresponding member function in the proxy class. A client application instantiates a proxy object at run time when it narrows an object reference.
Smart Proxies

For example, in the following code fragment, adapted from the familiar Messenger example from Chapter 3, a generic object reference (CORBA::Object) is narrowed to a more-specific type (Messenger), yielding a proxy to a Messenger object:

```cpp
// Obtain the Messenger’s object reference.
CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");

// Narrow it to type Messenger.
Messenger_var messenger = Messenger::_narrow(obj.in());
```

For more information on the role of proxies, see *Advanced CORBA Programming with C++*, 2.5.4.

To invoke an IDL operation or access an attribute on a target object, a client application calls the corresponding member function of the proxy class. This member function (via the client ORB) marshals the invocation into a request message and sends it to the server. The function then waits for the reply (unless it is a oneway or AMI function). When the reply arrives, the function (again via the client ORB) demarshals it and returns the reply values (the return value, plus the out and inout parameters) to the client application.

For each IDL interface, the TAO IDL compiler generates a proxy factory class that is used to construct the proxy object when an object reference is narrowed. The proxy class static member function _narrow() calls the factory’s create_proxy() member function, which in turn calls the proxy class constructor.

For most client applications, the default proxy class generated by the TAO IDL compiler is adequate. However, some applications can benefit from the customization of proxy behavior. TAO provides a way for an application developer to write a custom proxy class, known as a smart proxy class, that is used by the client application in lieu of the default proxy class. Smart proxies are not part of the CORBA specification, but since they are a client-side-only issue, they do not affect interoperability between different ORBs. Because the member functions of the smart proxy class have exactly the same signatures as those of the default class, no changes are required in the calls made by the client application on the proxy object. The added functionality of the smart proxy class is entirely transparent to the client application.
12.2 Smart Proxy Use Cases

The following use cases provide some insight into how smart proxies can be used.

- **Client-side caching**—The use of client-side caching is widespread in distributed computing. There are at least two advantages to caching information in the client:
  - Minimization of access time.
  - Minimization of the number of remote calls needed.

A common example is the caching of web pages by Internet browsers. By caching web pages as they are accessed, the browser allows repeated access to these pages without making repeated remote calls to the web server. This is advantageous to the user, who can access repeated pages more quickly, and to the web server, because its load is reduced.

The caching of pricing information can be quite useful in a purchasing system. When the client is only providing a price to a customer, it is often sufficient to provide this information from a client-side cache, rather than making a remote call. In the simplest case, the system can be set up to change prices only at specified times of the day. The smart proxy object can then cache pricing information and update it at those specified times. Remote calls are then only required during updates and purchases.

In a system that uses remote operations for time-consuming calculations that are often repeated, the smart proxy can cache the results of these operations and return them to the client application instead of making repeated remote calls.

None of the above cases requires changes to the target object’s interface. There are cases, however, where the efficiency of the system can be improved by adding additional operations to the interface. One example is the case of a client that frequently accesses multiple attributes of a target object. If an operation that accesses all of the attributes at once is added to the interface, the smart proxy can call this operation to get all the information at once, cache the information on the client-side, and provide the information to the client application through the individual attribute functions. If the operation is added to a derived interface, then the original interface need not be altered and can be provided to clients that do not use
smart proxies. (Another way to achieve similar behavior would be to redesign the interface to return a value type; see Chapter 11.)

- **Executing a sequence of operations**—If a client repeatedly makes the same sequence of operation invocations on one or more target object(s), the client application can be provided with a much simpler interface by building this functionality into a smart proxy and providing the client application with a single operation that encapsulates the entire sequence.

- **Choosing between target objects**—If more than one target object is available to serve a particular client request, the smart proxy can implement functionality that chooses which, among a number of possible target objects, to send the request to. One use of such functionality is to provide a form of load balancing by always sending the request to a lightly-loaded server.

- **Batch processing**—If a client application frequently makes a set of changes to a particular target object, a “batch” operation can be added to the target interface that makes all the changes at one time. The client will continue to make individual function calls on the smart proxy, but the smart proxy will cache these calls and combine them into a single request, thus reducing the number of remote calls. (Care must be taken if the smart proxy is used by multiple threads, of course.) If the batch operation is added to a derived interface, then the original interface need not be altered and can be provided to clients that do not use smart proxies.

- **Logical target object**—If a set of functions is to be provided to a client application by a number of target objects (with different interfaces), a smart-proxy class can be written to provide a single logical view of this collection of objects, whether they are legacy objects or not-yet-defined objects. The smart-proxy member functions contain the functionality required to call the correct operation on the correct target object. If new target operations are added later, only the smart proxy needs to be modified. The changes will be transparent to the client.

### 12.3 TAO’s Smart Proxy Framework

This section describes the classes that make up TAO’s smart-proxy framework, the responsibilities of each class, and how these classes interact. There are six C++ classes in the framework, but developers are only
responsible for writing two of them; the other four are provided in TAO or are generated by the IDL compiler. If you would like to get started writing and using smart proxy classes, you may skip now to 12.4 and refer to this section as needed.

Figure 12-1 shows the six classes that make up the TAO smart proxy framework.

**Figure 12-1 TAO’s Smart Proxy Framework**

The TAO_Smart_Proxy_Base class is part of the TAO source code (see $TAO_ROOT/tao/SmartProxies/Smart_Proxies.h). For each interface, the TAO IDL compiler generates a smart proxy base class that inherits from TAO_Smart_Proxy_Base. The smart proxy base classes are discussed in 12.3.3.

Invoking the TAO IDL compiler on an interface named MyInterface causes the classes MyInterface, MyInterface_out, and MyInterface_var to be generated. If the TAO IDL compiler is invoked with the -Gsp option on this same interface, three additional classes are generated that support smart proxies. They are:

- TAO_MyInterface_Proxy_Factory_Adapter
- TAO_MyInterface_Default_Proxy_Factory
Smart Proxies

• TAO_MyInterface_Smart_Proxy_Base

These three classes are discussed in 12.3.1 through 12.3.3.

Two classes are written by the application developer to provide application-specific behavior for the smart-proxy factory and the smart proxy. Your smart-proxy factory class will inherit from the default proxy-factory class generated by the TAO IDL compiler, and your smart proxy class will inherit from the smart-proxy base class, also generated by the IDL compiler. These classes are discussed in 12.4.

12.3.1 The Proxy Factory Adapter

The proxy-factory adapter uses a proxy factory to create proxies. It provides a consistent interface for both the default and developer-written proxy factories. It “adapts” the various proxy-factory interfaces to a common one. The proxy-factory adapter contains the following three member functions:

• register_proxy_factory()
• unregister_proxy_factory()
• create_proxy()

It also contains the proxy_factory_ member variable that stores a pointer to the proxy factory currently in use. The type of this pointer is the type of the default proxy factory. For the interface MyInterface, the type of this pointer is TAO_MyInterface_Default_Proxy_Factory.

The proxy-factory adapter class is instantiated as an ACE_Singleton object. For the interface MyInterface, it is defined by a typedef as the singleton class TAO_MyInterface_Proxy_Factory_Adapter. The three member functions of this class are discussed below.

• register_proxy_factory()

This function takes a pointer to a proxy factory as a parameter and stores it in the proxy_factory_ member variable. The default proxy-factory constructor calls this function to register itself with the adapter. A developer-written proxy-factory constructor will implicitly call this function because it inherits from the default factory, and thus implicitly calls the default factory constructor.

• unregister_proxy_factory()
This function deletes the currently-registered proxy-factory object, then sets the proxy_factory_ member variable to zero. It is called by the member function register_proxy_factory() prior to storing a new proxy factory pointer in the proxy_factory_ member variable. It is also called by the smart-proxy-base get_proxy() member function. It is necessary to unregister the currently-registered proxy-factory object before registering a new one to avoid getting into an infinite loop.

- create_proxy()

When this function is called, if the proxy_factory_ member variable points to a proxy factory, then that factory’s create_proxy() member function is called to create a new proxy object.

If the proxy_factory_ member variable is zero when this function is called (i.e., no factory object is currently registered), the default proxy-factory constructor is called. This constructor will register itself with the adapter by calling the adapter’s register_proxy_factory() member function. Once the registration process is completed, the proxy_factory_ member variable points to the default proxy factory. The factory’s create_proxy() function is then called.

12.3.2 The Default Proxy Factory
The default proxy factory serves both as the base class for developer-written proxy factories and as the default factory to be used in the absence of a developer-written factory. Its create_proxy() member function simply returns the proxy pointer that is passed to it as an argument. Because this function is called only by the proxy-factory adapter’s create_proxy() member function with an argument of a pointer to the default proxy, it always returns a pointer to the default proxy.

12.3.3 The Smart Proxy Base Classes
The TAO_Smart.Proxy_Base class is the base class for all TAO-IDL-compiler-generated smart-proxy base classes. Its constructor takes as an argument a pointer to the default proxy object that it stores in the member variable base_proxy_.

When the TAO IDL compiler is invoked with the -Gsp option, it generates a smart-proxy base class that inherits from both the default proxy class and the TAO_Smart.Proxy_Base class. For an interface named MyInterface, the
**Smart Proxies**

TAO\_MyInterface\_Smart\_Proxy\_Base class is generated. This class contains:

- The member variable `proxy_` that stores a pointer to the default proxy.
- The member function `get_proxy()` that returns the value of `proxy_`.
- Overridden member functions of the default proxy class that represent operations and attributes.

### 12.3.3.1 The overridden member functions

Each member function in the default proxy that represents an operation or attribute of the interface is overridden by a function in the smart-proxy base class generated by the TAO IDL compiler. The overriding function first calls the `get_proxy()` member function to get a pointer to the default proxy, then uses this pointer to call the overridden member function. For a default proxy function `op(arg)`, the body of the overriding function in the smart proxy base class is:

```
{
return this->get_proxy()->op(arg);
}
```

### 12.3.3.2 The `get_proxy()` function

The `get_proxy()` function returns a pointer to the default proxy. Because the value of the `proxy_` member variable is zero upon the first invocation of this function, a pointer to the default proxy must be obtained. This is accomplished by narrowing the pointer held by the `base_proxy_` variable, which is inherited from the TAO\_Smart\_Proxy\_Base class. For the `MyInterface` default base class, the definition of `get_proxy` is:

```cpp
MyInterface_ptr
TAO\_MyInterface\_Smart\_Proxy\_Base::get_proxy (void)
{
    // Obtain the real proxy stored in <base_proxy>
    if (CORBA::is_nil (this->proxy_.in()))
    {
        TAO\_MyInterface\_PROXY\_FACTORY\_ADAPTER::instance()->unregister_proxy_factory;
        this->proxy_ = ::MyInterface::unchecked_narrow(this->base_proxy_.in());
    }
    return this->proxy_.in();
}
```
12.3.4 An Overview of the Smart Proxy Creation Process

The key players in the smart proxy creation process are the proxy factory, the proxy-factory adapter, and the _unchecked_narrow() function. The proxy-factory adapter class and the default proxy-factory class are created along with the default proxy class (the standard proxy class that is always generated) when the TAO IDL compiler is invoked with the -Gsp option. A smart-proxy factory class that inherits from the default proxy-factory class is then written by the application developer.

12.3.4.1 The proxy factory and the proxy-factory adapter

The default proxy factory and the smart-proxy factory each contains a function named create_proxy(). The default proxy factory version of this function returns a pointer to the default proxy object, whereas the smart-proxy factory version of this function returns a pointer to a smart-proxy object. Neither function is called directly by the application to create a proxy. Instead, the proxy-factory adapter provides its own version of create_proxy() that chooses which factory to use when creating a proxy.

The proxy-factory adapter maintains a proxy-factory pointer as a data member. At the time a smart-proxy factory is constructed, a pointer to this factory is loaded into the adapter’s data member, replacing any previously-constructed factory. When the adapter’s create_proxy() function is called, it uses this pointer to call the create_proxy() function on the factory. If the pointer value is zero (because no smart-proxy factory has been constructed), the adapter will construct a default factory, then call create_proxy() on it.

12.3.4.2 The _unchecked_narrow() function

During the narrowing of an object reference, the _unchecked_narrow() function is called to provide a pointer to a proxy object. This function first loads a pointer to either the default proxy or a collocated object into the variable proxy. If the object reference is being narrowed to the MyInterface type, then the _unchecked_narrow() function will contain the following code:

```
return TAO_MyInterface_PROXY_FACTORY_ADAPTER::instance ()->create_proxy (proxy);
```
The proxy factory adapter singleton is instantiated and its create_proxy() function is called with default_proxy as its argument. If a smart proxy factory has already been constructed, the adapter calls create_proxy() on that factory. Otherwise, the default proxy factory will be constructed and its create_proxy() function will be called. In either case a pointer to the created proxy is ultimately returned by _unchecked_narrow().

Note that there is a singleton instance of a proxy factory adapter for each interface type.

12.4 Writing and Using Smart Proxy Classes

Using smart proxies in TAO involves the following steps:

- Compile the IDL interface(s) using the -Gsp option.
- Define the smart proxy class.
- Define the smart-proxy factory class.
- Instantiate a smart-proxy factory object.

12.4.1 The TAO IDL Compiler -Gsp Option

Compiling the IDL interface with the -Gsp option generates the additional classes needed to support the smart proxy feature. Of these classes, the two that are most relevant to using smart proxies are the smart-proxy base class and the default proxy factory. Your smart-proxy class will inherit from the generated smart-proxy base class. You will also need to create a smart-proxy factory class that inherits from the generated default proxy-factory class.

When the TAO IDL compiler is invoked on MyInterface with the -Gsp option, the generated smart-proxy base class and the default proxy-factory class are:

- TAO_MyInterface_Smart_Proxy_Base
- TAO_MyInterface_Default_Proxy_Factory

These classes are found in MyInterfaceC.h and MyInterfaceC.cpp along with the MyInterface proxy class.

The -Gsp option will automatically be added to the tao_idl options when your MPC project inherits from the smart_proxies base project. In fact, it
also adds the TAO_SmartProxies library to the set of libraries linked into the MPC project.

When a project involves multiple interfaces, some of which are used with smart proxies and some of which are not, less code will be generated if the interfaces that are not used with smart proxies are compiled separately without the -Gsp option. The following is an MPC example which shows how to use the -Gsp option with just the smart proxy related interfaces.

```plaintext
project: taoexe, smart_proxies {
    IDL_Files {
        with_smart_proxies.idl
    }
    IDL_Files {
        commandflags -= -Gsp
        without_smart_proxies.idl
    }
    Source_Files {
        main.cpp
        with_smart_proxiesC.cpp
        without_smart_proxiesC.cpp
    }
}
```

**Note** When an interface is compiled with the -Gsp option, the generated code uses the TAO_Smart_Proxy_Base class. Since this class is defined in the file $TAO_ROOT/tao/SmartProxies/Smart_Proxies.h, this file is automatically included by the generated code.

### 12.4.2 The Smart Proxy Class

Your smart-proxy class should inherit from the smart-proxy base class generated by the IDL compiler for your interface. You can name your smart-proxy class anything you want. For example, you might define a smart-proxy class called `Smart_MyInterface_Proxy` for the interface `MyInterface` as follows:

```plaintext
class Smart_MyInterface_Proxy : public TAO_MyInterface_Smart_Proxy_Base {
    ...);
```

Your smart-proxy class may override any of the operations defined in the base class. You only need to override the operations for which you want to provide
custom behavior in your smart-proxy class. The default behavior of these operations defined in the base class is to simply delegate to a standard proxy for the interface. The signatures of these operations follow the standard IDL-to-C++ mapping rules for interface operations.

12.4.3 The Smart Proxy Factory Class

Proxy objects are created by a proxy factory. You need to implement a factory to generate instances of your smart-proxy class. Your smart-proxy factory class should inherit from the proxy-factory class generated by the IDL compiler for your interface. You can name your proxy factory class anything you want. For example, you might define a smart proxy factory class called `Smart_MyInterface_Proxy_Factory` for the interface `MyInterface` as follows:

```cpp
class Smart_MyInterface_Proxy_Factory :
    public TAO_MyInterface_Default_Proxy_Factory
{
    public:
        virtual MyInterface_ptr create_proxy (MyInterface_ptr proxy);
        ...
};
```

Your factory’s `create_proxy()` function will be called by the proxy-factory adapter whenever a new proxy is needed (e.g., when an object reference is narrowed to the interface proxy type). Your implementation of the `create_proxy()` function should return a pointer to a proxy object. For example, it might create a new instance of your smart-proxy class or return a pointer to a singleton instance of your smart-proxy class.

You may provide any other functionality you need in your smart proxy factory class. For example, it may allocate resources it needs in its constructor and deallocate them in its destructor.

12.4.4 Instantiating a Smart Proxy Factory Object

When you want to use smart proxies, your client code should create an instance of your smart-proxy factory class before obtaining a proxy. When your smart-proxy factory is created, it will register itself with the proxy factory adapter, thereby replacing the adapter’s pointer to the default proxy factory. Then, when a proxy is needed (e.g., when an object reference is
narrowed to the interface proxy type), the proxy-factory adapter will call your proxy factory’s `create_proxy()` member function to obtain a proxy.

All you need to do in your client code is instantiate your proxy factory. Its base-class constructor contains the necessary logic to register your proxy factory with the proxy-factory adapter. For example, here is code to create and register a smart-proxy factory for the interface `MyInterface`:

```
// Create an instance of the smart proxy factory.
Smart_MyInterface_Proxy_Factory * factory =
    new Smart_MyInterface_Proxy_Factory();
// Obtain object references and narrow them as usual...
```

You should always instantiate your smart-proxy factory on the heap. The proxy-factory adapter assumes ownership of it and deletes it in `unregister_proxy_factory()`. *Do not attempt to delete it yourself.*

You can, of course, create more than one type of smart-proxy factory and/or smart proxy for a given interface. Simply instantiate the factory you want to use at run time. Only one proxy factory for a given interface type can be registered with that interface’s proxy-factory adapter at a time. Instantiating a new factory will cause the previous factory to be unregistered (and deleted) from the adapter before the new one is registered.

## 12.5 Linking Your Application

A Smart Proxy enabled application must be linked with the `TAO_SmartProxies` library. This library contains the `TAO_Smart_Proxy_Base` class from which all generated smart proxy classes inherit.

You can easily add this library (and the `tao_idl -Gsp` option) by adding the `smart_proxies` base project to your MPC project inheritance list. You will have to manually add the `TAO_SmartProxies` library to your list of libraries if you are not using MPC.
12.6  A Smart Proxy Example

The Messenger example introduced in Chapter 3 is extended here to use the smart proxy feature. We add a simple logging operation to the example through the introduction of a Logger interface. The Logger interface has one operation, log_message(), that writes a message to a log file. Our smart proxy’s send_message() operation calls log_message() to log information about usage of the Messenger both before and after it calls the default proxy’s send_message() member function. The complete source code for this example can be found in the TAO source code distribution in $TAO_ROOT/DevGuideExamples/SmartProxies.

12.6.1 The Messenger Interface

The Messenger interface is unchanged from the original example:

```c++
interface Messenger
{
    boolean send_message(in string user_name,
                          in string subject,
                          inout string message);
};
```

When the TAO IDL compiler is invoked on this interface with the -Gsp option, it generates the following classes in the files MessengerC.h and MessengerC.cpp:

- The class Messenger is the normal proxy class for the Messenger interface.
- The class TAO_Messenger_Smart_Proxy_Base inherits from both the Messenger class and the TAO_Smart_Proxy_Base class.
- The class TAO_Messenger_Default_Proxy.Factory creates default proxy objects and serves as the base for the Smart_Messenger.Proxy class that we use in this example.
- TAO_Messenger_PROXY_FACTORY_ADAPTER is the proxy factory adapter singleton.
12.6.2 Implementation of the Messenger Interface
Recall that the use of smart proxies is strictly a client-side issue; the server is unaware of the existence of smart proxies. Therefore, the implementation of the Messenger interface is unchanged from the original example and is not shown here.

12.6.3 The Logger Interface
The Logger interface is:

```cpp
// Logger.idl
type Logger
{
  boolean log_message (in string message);
};
```

12.6.4 Implementation of the Logger Interface
To implement the Logger interface, we first generate starter code by invoking the TAO IDL compiler on the Logger interface with the -GI option. Next, we rename the generated LoggerI.{h,cpp} files to Logger_i{h,cpp} and add our Logger’s implementation logic. In the code shown below, the added implementation code is shown in bold type.

```
#include "LoggerS.h"
#include <fstream>
class  Logger_i : public virtual POA_LOGGER
{
    public:
    Logger_i (void);
    virtual ~Logger_i (void);
    private:
    std::ofstream log_file; // Output file stream to which messages are logged.
    time_t log_time;        // Needed for creating a time stamp.
    char* log_time_string;  // The time stamp string.

    Logger_i() {
        log_file.open("log.txt", std::ios::app);
        log_time = time(NULL);
        log_time_string = ctime(&log_time);
    }

    virtual ~Logger_i() {
        log_file.close();
    }

    void log_message (const string& message) {
        log_file << log_time_string << " " << message << endl;
        log_time = time(NULL);
        log_time_string = ctime(&log_time);
    }

```

**Note** Smart proxies are not being used with the Logger interface, so the -Gsp option does not need to be used when compiling it since doing so will add unnecessary code to LoggerC.h and LoggerC.cpp.

The following code is from Logger_i.h:

```
#include "LoggerS.h"
#include <fstream>
class  Logger_i : public virtual POA_LOGGER
{
    public:
    Logger_i (void);
    virtual ~Logger_i (void);
    private:
    std::ofstream log_file; // Output file stream to which messages are logged.
    time_t log_time;        // Needed for creating a time stamp.
    char* log_time_string;  // The time stamp string.

    Logger_i() {
        log_file.open("log.txt", std::ios::app);
        log_time = time(NULL);
        log_time_string = ctime(&log_time);
    }

    virtual ~Logger_i() {
        log_file.close();
    }

    void log_message (const string& message) {
        log_file << log_time_string << " " << message << endl;
        log_time = time(NULL);
        log_time_string = ctime(&log_time);
    }

```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
virtual CORBA::Boolean log_message (const char * message)  
    throw(CORBA::SystemException);  
};

The following code is from the implementation of class Logger_i found in Logger_i.cpp:

#include "Logger_i.h"

Logger_i::Logger_i (void)  
{  
    log_file.open("Logger.txt");  
}

Logger_i::~Logger_i (void)  
{  
    log_file.close();  
}

CORBA::Boolean Logger_i::log_message (const char * message)  
    throw(CORBA::SystemException)  
{  
    ACE_OS::time(&log_time);  
    log_time_string = ACE_OS::ctime(&log_time);  
    // Replace carriage return with string delimiter.  
    log_time_string[24] = '\0';  
    log_file << log_time_string << " " << message << std::endl;  
    return 1;  
}

12.6.5 The Smart Proxy Factory
In this example, the smart-proxy factory class for the Messenger interface is named Smart_Messenger_Proxy_Factory. It inherits from the TAO_Messenger_Default_Proxy_Factory class that is generated by the IDL compiler. The factory class is defined in the file Smart_Messenger_Proxy.h as follows:

#include "MessengerC.h"
#include "LoggerC.h"

class Smart_Messenger_Proxy_Factory : public TAO_Messenger_Default_Proxy_Factory  
{  
    public:  
        Smart_Messenger_Proxy_Factory(CORBA::ORB_ptr orb);
virtual Messenger_ptr create_proxy ( 
    Messenger_ptr proxy);
private:
    Logger_var logger_; 
};

The constructor, found in the file Smart_Messenger_Proxy.cpp, is implemented as follows:

#include "Smart_Messenger_Proxy.h"
#include <iostream>
#include <stdexcept>

Smart_Messenger_Proxy_Factory::Smart_Messenger_Proxy_Factory( 
    CORBA::ORB_ptr orb)
{
    std::cout << "Creating smart proxy factory" << std::endl;
    // Convert the contents of the Logger.ior file to an object reference.
    CORBA::Object_var obj = orb->string_to_object("file://Logger.ior");
    if (CORBA::is_nil(obj.in())) {
        throw std::runtime_error ( 
            "Smart_Messenger_Proxy_Factory::CTOR: Nil Logger reference");
    }
    // Narrow the object reference to a Logger object reference.
    logger_ = Logger::_narrow(obj.in());
    if (CORBA::is_nil(logger_.in ())) {
        throw std::runtime_error ( 
            "Smart_Messenger_Proxy_Factory::CTOR: Not a Logger object reference");
    }
}

The following definition overrides the create_proxy() virtual member function from the base class:

Messenger_ptr 
Smart_Messenger_Proxy_Factory::create_proxy ( 
    Messenger_ptr proxy)
{
    Messenger_ptr smart_proxy = 0;
    if (CORBA::is_nil(proxy) == 0) 
        smart_proxy = new Smart_Messenger_Proxy(proxy, logger_.in());
    return smart_proxy;
}
The Smart Proxy Class
In this example, the smart-proxy class for the Messenger interface is named Smart_Messenger_Proxy. It inherits from the TAO_Messenger_Smart_Proxy_Base class that is generated by the IDL compiler. The Smart_Messenger_Proxy class is defined as follows:

```
#include "MessengerC.h"
#include "LoggerC.h"

class Smart_Messenger_Proxy : public TAO_Messenger_Smart_Proxy_Base
{
    public:
        Smart_Messenger_Proxy (Messenger_ptr proxy, Logger_ptr logger);
        virtual CORBA::Boolean send_message(
            const char* user_name,
            const char* subject,
            char*& message)
            throw (CORBA::SystemException);
    private:
        Logger_var logger_;
};
```

In the constructor, the Messenger proxy passed in is used to initialize the TAO_Smart_Proxy_Base base class. In addition, the object reference of the Logger object is duplicated and stored in a private data member of type Logger_var.

```
#include "Smart_Messenger_Proxy.h"
#include <iostream>

Smart_Messenger_Proxy::Smart_Messenger_Proxy(
    Messenger_ptr proxy, Logger_ptr logger)
    : TAO_Smart_Proxy_Base(proxy),
     logger_(Logger::_duplicate(logger))
{
    std::cout << "Creating smart proxy" << std::endl;
}
```

Now we override the send_message() operation in the smart proxy class. When the MessengerClient invokes send_message(), the smart proxy calls log_message() on the Logger proxy object both before and after it calls send_message() on the Messenger object. Note that it delegates the call to send_message() to its base class.
12.6 A Smart Proxy Example

CORBA::Boolean
Smart_Messenger_Proxy::send_message (  
    const char* user_name,  
    const char* subject,  
    char*& message)  
  throw (CORBA::SystemException)  
{
    logger_->log_message ("Before send_message()");
    CORBA::Boolean ret_val =
        TAO_Messenger_Smart_Proxy_Base::send_message (user_name, subject, message);
    logger_->log_message ("After send_message()");
    return ret_val;
}

12.6.7 The MessengerClient

The only changes needed in the original MessengerClient are to #include the smart proxy’s header file and to create an instance of the smart-proxy factory before obtaining and using a Messenger object reference. Here is the modified MessengerClient program:

```cpp
#include "MessengerC.h"
#include "Smart_Messenger_Proxy.h"
#include <iostream>
#include <stdexcept>

int main (int argc, char* argv[])  
{
  try {
    // Initialize the ORB
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Create a smart proxy factory. This will register it with the  
    // smart proxy factory adapter so it will be used to create  
    // Messenger proxies. Note that it is created on the heap, but is  
    // otherwise unused here.
    new Smart_Messenger_Proxy_Factory (orb.in());

    // Convert the contents of the Messenger.ior file to an object reference.  
    CORBA::Object_var obj = orb->string_to_object ("file://Messenger.ior");
    if (CORBA::is_nil(obj.in())) {
      std::cerr << "Nil Messenger reference" << std::endl;
      return 1;
    }

    // Narrow the object reference to a Messenger object reference.  
    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (CORBA::is_nil(messenger.in())) {  
```
std::cerr << "Not a Messenger object reference" << std::endl;
    return 1;
}

// Create a message and send it to the Messenger.
CORBA::String_var message = CORBA::string_dup("Hello!");
messenger->send_message("TAO User", "TAO Test", message.inout());
std::cout << "Message was sent" << std::endl;

// Release resources.
orb->destroy();
}
catch (CORBA::Exception& e) {
    std::cerr << "Caught a CORBA exception: " << e << std::endl;
    return 1;
}
catch (std::exception& ex) {
    std::cerr << ex.what() << std::endl;
    return 1;
}

return 0;
}

12.6.8 The MessengerServer
Recall that the use of smart proxies on the client side is completely transparent to the server side. Thus, the MessengerServer is unchanged from the original example and is not shown here.

12.6.9 The LoggerServer
The LoggerServer is very simple. In fact, it is identical to the MessengerServer except that every instance of the name “[Mm]essenger” in the MessengerServer is replaced with the name “[Ll]ogger” in the LoggerServer.

#include "Logger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get a reference to Root POA.
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

// Activate the POA manager.
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Create a Logger_i servant.
Logger_i logger_servant;

// Register the servant with the RootPOA, obtain its object reference,
// stringify it, and write it to a file.
PortableServer::ObjectId_var oid = poa->activate_object(&logger_servant);
CORBA::Object_var logger_obj = poa->id_to_reference(oid.in());
CORBA::String_var str = orb->object_to_string(logger_obj.in());
std::ofstream iorFile("Logger.ior");
iorFile << str.in() << std::endl;
iorFile.close();
std::cout << "IOR written to file Logger.ior" << std::endl;

// Accept requests from clients.
orb->run();

// Release resources.
orb->destroy();
}
catch (CORBA::Exception&) {
  std::cerr << "Caught a CORBA exception." << std::endl;
  return 1;
}
return 0;

12.6.10 Running the Programs
Compile the LoggerServer, MessengerServer, and MessengerClient programs. Then, start the LoggerServer and MessengerServer before running the MessengerClient.

12.6.11 The Output
When you run the MessengerServer, you will see the following output:

IOR written to file Messenger.ior
When you run the `LoggerServer`, you will see the following output:

IOR written to file Logger.ior

When you run the `MessengerClient`, you will see the following output from the client:

Creating smart proxy factory
Creating smart proxy
Message was sent

and the following output from the `MessengerServer`:

Message from: TAO User
Subject:      TAO Test
Message:      Hello!

In addition, the `Logger.txt` file will contain something similar to the following:

Mon Sep 15 12:07:57 2003 Before send_message()
Mon Sep 15 12:07:57 2003 After send_message()
CHAPTER 13

Local Interfaces

13.1 Introduction

TAO implements the concept of “local” interfaces as first introduced in the CORBA Component Model (CCM) specification (OMG Document formal/02-06-65, 4.1) and added to the CORBA 2.6 specification (OMG Document formal/01-12-35, 3.7.6). The above specifications define a standard set of behaviors for locality-constrained objects (cannot be invoked externally or remotely). Previously, pseudo-IDL (PIDL) was used to define operations that the ORB supported but were intended only for the local application. PIDL has been deprecated in favor of standard IDL with interfaces specified as local.

Note

The file extension PIDL is still used on many files. Many of these files are now true IDL. Some are still pseudo-IDL.

With the inclusion of local interfaces, specifications can avoid creation of special pseudo-objects that are never called remotely, such as the POA and ORB interfaces. Local interfaces permit you to define portable local classes
Local Interfaces

that are independent of any ORB-specific optimizations for local access (e.g., interceptors are not called, POA policies are not honored). Local objects are important for server-only components. They help improve performance and minimize memory footprint by avoiding the overhead of ORB mediation. In addition, the use of a local interface permits the consistent disposition of objects because locality-constrained objects and remote objects share a uniform method of definition (IDL).

A local interface is specified in the IDL with the keyword local before the keyword interface. An instance of a local interface is a local object. A standard IDL interface is an unconstrained interface. Local interfaces and objects differ from unconstrained interfaces and objects in several ways:

• The ORB does not mediate any invocation on a local object. Implementations of local interfaces are responsible for providing the parameter copy semantics that clients expect.

• A local interface may inherit from other local or unconstrained interfaces. However, an unconstrained interface may not inherit from a local interface.

• Local types cannot be marshaled, externalized, or invoked from another process. References to local objects cannot be converted to strings through CORBA::ORB::object_to_string(). An attempt to do so raises a CORBA::MARSHAL system exception with the minor code set to 4.

• A local object is valid only in the process in which it is instantiated.

• There is no concept of an “object id” for a local object. Its identity is implementation specific (e.g., pointer or reference).

• Neither the Dynamic Invocation Interface (DII) nor Asynchronous Method Invocation (AMI) is supported on local objects.

• Instances of certain local objects that are part of the OMG specification (e.g., POA) are obtained via resolve_initial_references().

13.2 C++ Mapping for LocalObject

A locality-constrained class is derived from both CORBA::LocalObject and the class mapping the interface. The class CORBA::LocalObject is used as a base class for locality-constrained implementations. It is derived from
13.3 Changing Existing Interfaces to Local Interfaces

CORBA::Object. The CORBA::LocalObject class implements the following CORBA::Object pseudo-operations. The operations throw the CORBA::NO_IMPLEMENT exception.

- is_a()
- get_interface()
- get_domain_managers()
- get_policy()
- get_cached_policy()
- set_policy_overrides()
- get_policy_overrides()
- validate_connection()

The CORBA::LocalObject class provides implementations of the following CORBA::Object pseudo-operations:

- non_existent() — always returns FALSE.
- hash() — returns a consistent hash value for the lifetime of the object.
- is_equivalent() — returns TRUE if the references refer to the same CORBA::LocalObject implementation.

13.3 Changing Existing Interfaces to Local Interfaces

With the adoption of local interfaces, the specification changed the following into local interfaces:

- CORBA::Current
- All the interfaces in the DynamicAny module.
- All the interfaces in the PortableServer module.

The following CORBA messaging interfaces have become local interfaces:

- CORBA::PolicyManager
- CORBA::PolicyCurrent
- CORBA::Pollable
Local Interfaces

- CORBA::DIIPollable
- CORBA::PollableSet
- All interfaces in the Messaging module that inherit from CORBA::Policy

13.4 Example: ServantLocator

The POA determines which servant is associated with a particular request by calling upon the application's Servant Manager. An application registers a Servant Manager with a POA. There are two types of Servant Managers, depending upon whether or not the POA retains the associations of objects to servants in its Active Object Map. The ServantRetentionPolicy is used to determine if the POA retains servant activations in the Active Object Map (RETAIN) or does not (NON_RETAIN). When this policy is set to RETAIN, the Servant Manager must activate the servant associated with the object and implement the PortableServer::ServantActivator interface. When the ServantRetentionPolicy is set to NON_RETAIN, the Servant Manager must locate the servant associated with the object and must implement the PortableServer::ServantLocator interface.

The following example modifies the Messenger example from Chapter 3. A Servant Locator is used to locate the servant associated with the Messenger object when a request is invoked on this object. The complete source code for this example can be found in the TAO source code distribution in $TAO_ROOT/DevGuideExamples/LocalObjects/ServantLocator.

13.4.1 The Messenger Locator Implementation

The following implementation of the Messenger_Locator_i class will find the Messenger servant that incarnates the target object of the request. The PortableServer::ServantLocator interface provides two operations: preinvoke() and postinvoke(). The preinvoke() operation is invoked by the POA to bind a servant to the target CORBA object when an incoming request is received. The postinvoke() operation is invoked by the POA to release the servant once the request has been fulfilled. The following code shows the definition of the Messenger_Locator_i implementation class:

```c++
#include <tao/corba.h>
```
#include <tao/PortableServer/PortableServer.h>

class Messenger_Locator_i :
  public PortableServer::ServantLocator,
  public CORBA::LocalObject
{
  public:
  Messenger_Locator_i ();

  // Preinvoke function
  virtual PortableServer::Servant preinvoke (
    const PortableServer::ObjectId& oid,
    PortableServer::POA_ptr poa,
    const char* operation,
    void*& cookie)
  throw (CORBA::SystemException, PortableServer::ForwardRequest);

  // Postinvoke function
  virtual void postinvoke (
    const PortableServer::ObjectId& oid,
    PortableServer::POA_ptr poa,
    const char* operation,
    void* cookie,
    PortableServer::Servant servant)
  throw (CORBA::SystemException);
};

The Messenger_Locator_i class inherits from both its skeleton class, PortableServer::ServantLocator, and CORBA::LocalObject. To implement a PortableServer::ServantLocator, you must inherit from both PortableServer::ServantLocator and CORBA::LocalObject.

The preinvoke() operation converts the ObjectId to a string. If the ID is valid, a new servant is created and returned. The cookie allows the ServantLocator to pass data from an invocation of preinvoke() to the corresponding postinvoke() invocation. The following code shows the implementation of the preinvoke() operation:

PortableServer::Servant
Messenger_Locator_i::preinvoke (
  const PortableServer::ObjectId& oid,
  PortableServer::POA_ptr, poa,
  const char*,
  void*& cookie)
  throw (CORBA::SystemException, PortableServer::ForwardRequest)
{
// Get the ObjectId in string format.
CORBA::String_var oid_str = PortableServer::ObjectId_to_string (oid);

std::cout << "preinvoke called..." << oid_str << std::endl;

// Check if the ObjectId is valid.
ACE_CString cstr(oid_str.in());
if (cstr == "Messenger") {
    // Create the required servant
    PortableServer::ServantBase_var servant = new Messenger_i ();

    // Set a flag so that we know to delete it in postinvoke().
    cookie = (void *)1;

    return servant._retn();
} else {
    throw CORBA::OBJECT_NOT_EXIST ();
}

In the postinvoke() operation, the servant is simply destroyed:

void
Messenger_Locator_i::postinvoke (const PortableServer::ObjectId&,
        PortableServer::POA_ptr,
        const char*,
        void* cookie,
        PortableServer::Servant servant)
        throw (CORBA::SystemException)
{
    std::cout << "postinvoke called..." << std::endl;

    // Delete the servant as it is no longer needed.
    if (cookie != 0) {
        delete servant;
    }
}

13.4.2 The Server Implementation
We now modify the original MessengerServer.cpp to use our newly-created Messenger_Locator_i class.

First, we replace #include "Messenger_i.h" with #include "MessengerLocator_i.h".
We create a new POA (the childPOA) with the USE_SERVANT_MANAGER value for the RequestProcessingPolicy and NON_RETAIN for the ServantRetentionPolicy as follows:

```plaintext
// Create the policies and assign them for the child POA
CORBA::PolicyList policies (3);
policies.length (3);

policies [0] = rootPOA->create_id_assignment_policy (PortableServer::USER_ID);
policies [1] = rootPOA->create_request_processing_policy (PortableServer::USE_SERVANT_MANAGER);
policies [2] = rootPOA->create_servant_retention_policy (PortableServer::NON_RETAIN);

// Create the POA with these policies
PortableServer::POA_var childPOA =
    rootPOA->create_POA ("childPOA", mgr.in(), policies);

// Destroy the policy objects
for (CORBA::ULong i = 0; i != policies.length(); ++i)
    policies [i]->destroy();
```

Now that we have a POA that can support Servant Managers, we need to create an instance of a ServantLocator and set it as the Servant Manager for the childPOA:

```plaintext
// Create our Messenger’s ServantLocator.
PortableServer::ServantLocator_var locator = new Messenger_Locator_i;

// Set the Servant Manager with the childPOA.
childPOA->set_servant_manager (locator.in());
```

The ServantLocator is now registered with the childPOA. We now create a Messenger reference via the childPOA with the user-defined ObjectId:

```plaintext
// Get the object id for the user-created ID in the childPOA
PortableServer::ObjectId_var child_oid =
    PortableServer::string_to_ObjectId ("Messenger");

// Create the object without creating a servant.
CORBA::Object_var messenger_obj =
    childPOA->create_reference_with_id (child_oid.in(), ::_tc_Messenger->id());
```

As before, we write the object reference as a stringified IOR to the file Messenger.ior:
// Put the object reference into an IOR string
CORBA::String_var str = orb->object_to_string (messenger_obj.in());

// Write the IOR string to a file
std::ofstream iorFile ("Messenger.ior");
iorFile << str.in ();
iorFile.close ();
std::cout << "IOR written to the file Messenger.ior." << std::endl;

We then handle incoming requests from clients:

// Accept requests from clients.
orb->run();

13.4.3 An Example Using a Local Object
In this section, we discuss a rather contrived example, where the Messenger interface itself is defined as local:

// Messenger.idl
local interface Messenger
{
    boolean send_message (  
        in string user_name,  
        in string subject,  
        inout string message);
};

The Messenger interface is now locality-constrained. The ORB will not mediate requests to instances of the Messenger class. The overhead associated with a call to send_message() is on the order of one virtual function call, making possible performance improvements of large magnitude over collocated operation invocations on normal CORBA objects.

The complete source code for this example can be found in the TAO 1.4a source code distribution:
$TAO_ROOT/DevGuideExamples/LocalObjects/Messenger.

The definition of the Messenger_i class becomes:

// Class Messenger_i
class Messenger_i :
    public virtual Messenger,
    public virtual CORBA::LocalObject
{ public:
    //Constructor
    Messenger_i (void);

    //Destructor
    virtual ~Messenger_i (void);

    virtual CORBA::Boolean send_message (const char* user_name,
                                         const char* subject,
                                         char*& message
    )
    ACE_THROW_SPEC ((CORBA::SystemException));
};

Note

The TAO IDL compiler’s -GI option should not be used to automatically
generate implementation-class starter code for IDL files containing local
interfaces.

The definition of the Messenger_i class is identical to the earlier example
with the unconstrained interface:

#include  "Messenger_i.h"

// Implementation skeleton constructor
Messenger_i::Messenger_i (void) {
}

// Implementation skeleton destructor
Messenger_i::~Messenger_i (void) {
}

CORBA::Boolean Messenger_i::send_message (const char* user_name,
                                          const char* subject,
                                          char*& message
)  
ACE_THROW_SPEC ((CORBA::SystemException))
Local Interfaces

```cpp
// my implementation
std::cout << "Message from: " << user_name << std::endl;
std::cout << "Subject:      " << subject << std::endl;
std::cout << "Message:      " << message << std::endl;
CORBA::string_free(message);
message = CORBA::string_dup("Thanks for the message.");
return 1;
}

The server and client are, by definition, collocated for this example. The code below shows the construction and use of a Messenger object:

```c
#include "Messenger_i.h"
#include <iostream>

int main (int argc, char * argv[])
{
  try {
    // Construct a Messenger object and use it "as if" it's a corba object.
    // Put it into CORBA object reference
    // comparable to activation, narrow, etc.
    Messenger_var messenger (new Messenger_i);

    // Send a message to the Messenger object.
    CORBA::String_var message = CORBA::string_dup ("Hello!");
messenger->send_message("TAO User", "TAO Test", message.inout());

    // Print the Messenger's reply.
    std::cout << "Reply: " << message.in() << std::endl;
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught CORBA::Exception : " << ex << std::endl;
    return 1;
  }

  return 0;
}

Recall that if you initialize a _var reference with a _ptr reference, as shown here:

        Messenger_var messenger (new Messenger_i);
```
the \_var takes ownership (without incrementing the reference count on the proxy) and eventually calls \texttt{CORBA::release()} on the underlying \_ptr. The usage of this \_var is equivalent to the \_var of unconstrained objects.

### 13.4.4 Reference Counting and Local Objects

Local objects can use reference counting to manage their life cycles just as unconstrained CORBA objects. The class \texttt{CORBA::LocalObject} defines two virtual functions:

- \_add\_ref(): This member function is called when the reference is duplicated. A default implementation is provided that does nothing. A derived implementation may use this operation to maintain a reference count.

- \_remove\_ref(): This member function is called when the reference is released. A default implementation is provided that does nothing. A derived implementation may use this operation to maintain a reference count, and delete the object when the count becomes zero.

The fact that the default implementations for these two operations do nothing implies that the local objects are responsible for managing their own life cycles. TAO developers who wish to perform thread-safe reference counting on local object instances can inherit their local object implementation classes from \texttt{TAO\_Local\_RefCounted\_Object} instead of \texttt{CORBA::LocalObject}:

```cpp
class Messenger_i :  
    public virtual Messenger,  
    public virtual TAO\_Local\_RefCounted\_Object  
{  
    // Remainder of Messenger\_i unchanged.  
};
```

---

**Note**  
Implementations that use the \texttt{TAO\_Local\_RefCounted\_Object} mix-in class are not portable to other ORBs.
14.1 Introduction

When a client sends a request to a server, the object key portion of the interoperable object reference (IOR) of the target object is included in the GIOP request message header. The server's ORB uses the object key to demultiplex the request to the appropriate target servant. Typically, the ORB looks up the POA via the POA name portion of the object key, then the POA locates the servant via the ObjectId portion of the object key.

The CORBA specification defines the corbaloc object URL scheme to provide for convenient exchange of human-readable IORs. Here is an example of a corbaloc object URL:

corbaloc:iiop:malory:5555/ObjectKeyString

See 19.8.13 for more information on corbaloc object URLs.

Unfortunately, object keys are often not human-readable, making it difficult for users to construct simple corbaloc object URLs such as the example above. Users need a way to specify a simple object key string in corbaloc
object URLs and servers need a way to map these simple object key strings to target objects. This chapter describes TAO’s IOR Table feature that allows a server to map simple object key strings to the actual IORs of target objects.

## 14.2 IOR Table

The IOR Table is a TAO-specific local object that allows a server to expose an object reference as a simple corbaloc object URL by mapping simple object key strings to stringified IORs. The IOR Table provides a form of indirect binding similar to that provided by the TAO Implementation Repository (ImR), and can be used by any server. Many TAO services, such as the Naming Service, Notification Service, and Implementation Repository support such indirect binding.

The TAO IORTable::Table local interface is defined in $TAO_ROOT/tao/IORTable/IORTable.pidl. It provides simple bind(), rebind(), and unbind() operations. A portion of the interface definition is shown here:

```plaintext
module IORTable
{
 exception AlreadyBound {};
 exception NotFound {};

 local interface Table
{
   void bind (in string object_key, in string IOR)
     raises (AlreadyBound);
   void rebind (in string object_key, in string IOR);
   void unbind (in string object_key)
     raises (NotFound);
};
}
```

Your server can access the ORB’s IOR Table by calling resolve_initial_references("IORTable"). You then use the bind() operation to bind a simple object key string to a stringified IOR, as shown in this example:

```plaintext
#include <tao/IORTable/IORTable.h>
...

// Initialize the ORB, get the RootPOA, activate servants, and
// generate object references as usual (not shown).
```
// Stringify your object references for binding in the IOR Table.
CORBA::String_var messenger_str1 = orb->object_to_string (ior1.in());
CORBA::String_var messenger_str2 = orb->object_to_string (ior2.in());

// Get the IORTable from the ORB.
CORBA::Object_var table_obj = orb->resolve_initial_references("IORTable");
IORTable::Table_var table = IORTable::Table::_narrow(table_obj.in());

// Bind your objects to simple object key strings.
tbl->bind("Messenger1", messenger_str1);
tbl->bind("Messenger2", messenger_str2);

The simple object key strings can also indicate the persistent POA name in
which the object is activated (if not the Root POA), as shown here:

tbl->bind("MessengerService/Messenger1", messenger_str1);
tbl->bind("MessengerService/Messenger2", messenger_str2);

Now, if clients know the simple object key string of the object they want to
access, and the endpoint on which the server is listening, they can use simple
corbaloc ObjectURLs such as the following:

corbaloc:iiop:malory:5555/Messenger1

The client would normally provide this ObjectURL to the ORB’s
string_to_object() operation, as follows:

CORBA::Object_var obj =
    orb->string_to_object("corbaloc:iiop:malory:5555/Messenger1");
Messenger_var myobj = Messenger::_narrow(obj.in());

In this case, we assume the server was stared using and endpoint specification
such as:

-ORBListenEndpoints iiop://malory:5555

and that the object reference bound to "Messenger1" in the server
implements the Messenger interface.

Typically, the first request sent to the object is an invocation of the implicit
is_a() operation that results from a call to the static narrow() function as
shown in the above example. When the request reaches the server’s ORB, it
matches the simple object key string from the ObjectURL to a binding in the
IORTable. It then returns a LOCATION_FORWARD reply containing the IOR that was bound to the simple object key string to the client. The client’s ORB automatically reinvokes the request on this returned IOR. This second invocation will be demultiplexed to the target object as usual. The client will continue to use the returned IOR for subsequent requests.

The IORs bound in the IORTable can be of any valid format accepted by string_to_object(), including IOR:, corbaloc, and file://. (See 19.8.13 for more information on valid IOR formats.) Clients will be forwarded to the location indicated by the returned IOR using a LOCATION_FORWARD reply. Note that a client could be forwarded to a different server than that from which it received the LOCATION_FORWARD reply, depending upon the profiles in the returned IOR.

14.3 Locator

In addition to mapping simple object key strings to IORs as described above, you can also interact with the IORTable by registering a Locator object. A Locator is a user-defined local object that the IORTable can use to find objects. If the table is unable to find a binding for the object key string in an incoming request, it will call a function in your Locator object, passing the object key string. The Locator can use any user-defined mechanism to return a valid IOR. For example, you could use this mechanism to support case-insensitive IORTable lookup.

The IORTable::Locator local interface is defined in $TAO_ROOT/tao/IORTable/IORTable.pidl. It provides a simple locate() operation that takes an object key string and returns a stringified IOR. You register your Locator with the IORTable via the table’s set_locator() operation. These interfaces are shown here:

```plaintext
module IORTable
{
    local interface Locator;  // forward declaration

    exception NotFound {};

    local interface Table
    {
        // Other operations shown previously...
    }

    // Other operations shown previously...
}
```
To use a Locator with the IORTTable, simply create a class that derives from IORTTable::Locator and CORBA::LocalObject and override the locate() method to return a stringified IOR for the given object key string. For example:

```c++
#include <tao/IORTTable/IORTTable.h>

class MessengerLocator :
  public IORTTable::Locator,
  public CORBA::LocalObject
{
  // Our Locator will use an internal map of key strings to stringified IORs.
  std::map<std::string, std::string> map_;

  public:
  MessengerLocator ()
  {
    // Initialize the map of keys and stringified IORs (not shown).
  }
  // Override IORTTable::Locator::locate().
  virtual char* locate (const char* key)
  {
    return map_[key];
  }
};
```

For more information on implementing local interfaces, see Chapter 13.

Register your locator with the IORTTable by calling set_locator() in your server’s startup code:

```c++
class MessengerServer
{
  MessengerLocator locator_;
  public:
  void run();
};
```
void MessengerServer::run()
{
    // Initialize the ORB, get the RootPOA, activate servants, and
    // generate object references as usual (not shown).

    // Get the IORTable from the ORB.
    CORBA::Object_var obj = orb->resolve_initial_references("IORTable");
    IORTable::Table_var tbl = IORTable::Table::_narrow(obj.in());

    // Bind some objects to simple object key strings (as before).
    tbl->bind("Messenger1", messenger_str1);
    tbl->bind("Messenger2", messenger_str2);

    // Use our Locator to resolve any object key string not bound in the table.
    tbl->set_locator(&locator_);

    // ...
}
CHAPTER 15

Using Pluggable Protocols

15.1 Introduction

With the adoption of CORBA version 2.0, the OMG defined a wire protocol for inter-ORB communication. This protocol is defined as a messaging layer, known as the General Inter-ORB Protocol (GIOP), combined with at least one transmission protocol, TCP/IP. The mapping of GIOP on to TCP/IP is known as the Internet Inter-ORB Protocol (IIOP). The CORBA specification also defines a protocol known as the DCE Environment Specific Inter-ORB Protocol (DCE ESIOP). To be compliant with the standard, all ORB vendors must supply an implementation of IIOP. Vendors may additionally supply ESIOPs to enable inter-ORB communication using varied IPC mechanisms.

Like many CORBA implementations, TAO provides several ESIOPs. This chapter describes how to use and configure the different protocols that are part of the TAO 1.4a release. TAO additionally provides a programming framework so that others may produce alternative inter-ORB protocols. Chapter 16 describes how to develop your own ESIOP and integrate it with TAO.
Note: What has been historically referred to as a “pluggable protocol” in TAO is really a pluggable transport. TAO does not currently support pluggable messaging (i.e., only GIOP is supported), nor is it possible to choose alternative marshaling schemes (i.e., only CDR is supported). However, for historical reasons, all of the pluggable transports described in this chapter are referred to as pluggable protocols.

15.2 Protocol Introduction

Each inter-ORB protocol consists of three basic components: messaging, marshaling, and transport. A given ESIOP consists of concrete definitions of all three of these. As an example, the IIOP protocol, as defined by the OMG, uses GIOP messaging, CDR marshaling, and the TCP/IP transport. Chapter 15 of the CORBA 3.0.3 specification provides details of all the layers of inter-ORB protocols. Further insight into these protocols may be found in IIOP Complete: Understanding CORBA and Middleware Interoperability by William Ruh, Thomas Herron, and Paul Klinker (1999).

15.2.1 Messaging

All of TAO's protocols use the General Inter-ORB Protocol (GIOP) as their messaging layer. TAO does not currently provide a way to replace the messaging layer.

The CORBA 3.0.3 Core specification (OMG Document formal/04-03-12) defines GIOP as a description of the messages that clients and servers exchange, and many other details of their interaction. The messages are defined as IDL structures in Chapter 15 of the CORBA specification.

Although GIOP is defined as transport-independent, it does make several transport assumptions that restrict the use of transports. The main assumptions are that the transport:

- is connection-oriented.
- connection initiation is similar to TCP (the server can publish a network address at which it’s listening in an IOR).
- is reliable according to the TCP definition (bytes are delivered in order, at most once, and are acknowledged).
• can be viewed as a byte stream (no message size limitations, fragmentation, or alignments).

• provides notification of disorderly connection loss.

This does not mean that the transport must directly adhere to these features, just that the transport layer shows this behavior to the messaging layer.

Many of TAO’s protocols can also be configured to use GIOP Lite, which is a TAO-Specific version of GIOP that provides efficiency advantages in some applications. GIOP Lite removes several fields from GIOP messages at a cost of portability and some flexibility. See 15.7 and 20.6.12 for more information on using GIOP Lite.

15.2.2 Marshaling

The marshaling layer defines the transformation of inter-ORB messages into and out of a transmission (wire) format. The Common Data Representation (CDR), performs this role for all of TAO’s provided protocols. TAO does not currently provide a way to replace the marshaling layer. CDR is able to account for variable byte ordering on different hardware architectures, efficiently align data types, and transfer any data type defined in IDL.

15.2.3 Transport

The most common CORBA protocol, IIOP, is GIOP mapped to TCP/IP. The transport layer defines connection establishment and inter-ORB message transmission. The transport layer also defines how connection “endpoints” are represented in the profiles of Interoperable Object References (IORs).

When protocols utilize low-level transports that do not support the messaging layer’s transport assumptions, the transport layer of the protocol must simulate the behavior that meets these assumptions. For example, transports that are not connection oriented (like UDP or shared memory) must simulate connections.

TAO’s pluggable protocol framework provides an easy way to replace the transport layer. When you write a pluggable protocol, you are writing a new transport layer. Future work may provide for replacement of the other layers.
15.3 Protocols Provided with TAO

The rest of this chapter discusses the inter-ORB protocols supplied with TAO, how to use them in your application, and how to control their behavior. The main motivator for using these protocols is typically higher performance, although security or reduced bandwidth consumption may also motivate their use. Since CORBA allows multiple profiles in an object reference, it is possible to access the same CORBA object using more than one protocol. The protocols provided with TAO 1.4a are shown in Table 15-1.

Table 15-1 Available Pluggable Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIOP</td>
<td>(default) Internet Inter-ORB Protocol. This protocol is required for CORBA 2.0+ compliance (found in -lTAO).</td>
</tr>
<tr>
<td>IIOP_Lite</td>
<td>A variation of IIOP that uses a smaller, proprietary format for inter-ORB message headers (in -lTAO).</td>
</tr>
<tr>
<td>UIOP</td>
<td>Local IPC (i.e., Unix domain) socket-based transport protocol (found in -lTAO_Strategies).</td>
</tr>
<tr>
<td>UIOP_Lite</td>
<td>A variation of UIOP that uses a smaller, proprietary, format for inter-ORB message headers (found in -lTAO_Strategies).</td>
</tr>
<tr>
<td>SHMIOP</td>
<td>Shared memory transport protocol (found in -lTAO_Strategies).</td>
</tr>
<tr>
<td>DIOP</td>
<td>Datagram (UDP) transport protocol (found in -lTAO_Strategies).</td>
</tr>
<tr>
<td>SSLIOP</td>
<td>IIOP over Secure Sockets Layer (SSL) session protocol (found in -lTAO_SSLIOP). This protocol requires the freely available OpenSSL library.</td>
</tr>
<tr>
<td>SCIOP</td>
<td>SCTP Inter-ORB Protocol (found in -lTAO_Strategies). An SCTP-based protocol, currently only for UNIX platforms.</td>
</tr>
<tr>
<td>HTIOP</td>
<td>HTTP Tunneling Inter-ORB Protocol (requires -lTAO_HTIOP and -lACE_HTBP). An asymmetric protocol for communication across a firewall using HTTP tunnels.</td>
</tr>
<tr>
<td>MIOP/UIPMC</td>
<td>Multicast Inter-ORB Protocol (MIOP) Unreliable IP Multicast (UIPMC) protocol (found in -lTAO_PortableGroup).</td>
</tr>
</tbody>
</table>

The following sections describe each of these protocols in detail, including the exact steps needed to use the protocol. In general, the steps needed to use each existing protocol are:

1. Build the library supplying the protocol.
15.4 Building the Protocol Libraries

2. Load and initialize the protocol. TAO’s pluggable protocols are all defined as dynamically-loadable services. A factory object creates the components that collaborate to implement the protocol.

3. Identify the protocol to the ORB. This allows the ORB to request the creation of protocol components from each of the identified factories.

4. Supply any required endpoint information to be used by the ORB initialization run-time code.

15.4 Building the Protocol Libraries

As noted in Table 15-1, the base TAO library includes IIOP. The library also contains IIOP_Lite, but that protocol is not used by the ORB unless specifically declared.

The SSLIOp components are built in a separate ORB service library, which depends on a third-party security package, as described in 15.11.

The UIPMC protocol is included as part of TAO’s Portable Group implementation in the TAO_PortableGroup library. This library is located in $TAO_ROOT/orbsvcs/orbsvcs/PortableGroup.

The HTIOP protocol depends upon the HTBP library for the communication between the inside and outside peer through a firewall. This library is located in $ACE_ROOT/protocols/ace/HTBP. The HTIOP library is located in $TAO_ROOT/orbsvcs/orbsvcs/HTIOP.

The remaining protocols are all available as part of the TAO_Strategies library. This library is located in $TAO_ROOT/tao/Strategies. The default makefile or project file for TAO will build the strategy library in the course of building the base TAO library. Separate make and project files are available if you wish to build the strategy library individually.

15.5 Loading Pluggable Protocols

To use a protocol in TAO, you must declare its protocol factory by using the ACE Service Configurator (see 20.6.12). This factory instantiates such things as the acceptor and connector responsible for establishing connections.

Declaring a protocol factory is done through options passed to the ORB’s resource factory (Chapter 20) in the service configurator’s (Chapter 18)
configuration file. The resource factory option used is
-ORBProtocolFactory, described in 20.6.12 Further details about
individual protocol factories are given below.

Protocol factories are dynamically loadable. If you wish to use a non-default
protocol, or one which is loaded automatically as a result of using a specific
resource factory, you must direct the service configurator to load the protocol
factory object, in addition to supplying the argument to the resource factory. A
dynamic directive, as shown in 18.3.2, is used to load the protocol factory
code. Specific directives for loading the various protocol factories are shown
below.

Pay close attention to the syntax for the dynamic directive. It is tricky, and
incorrect syntax is the cause of most problems encountered when dynamically
loading protocols.

15.6 IIOP

The Internet Inter-ORB Protocol is the most widely used CORBA protocol,
and is required by the CORBA standard. IIOP specifies how GIOP messages
are sent, such that invocations may be made from client to server over a
TCP/IP network. IIOP, along with TAO’s other protocols, is implemented as a
pluggable protocol. This enables IIOP to be combined with other protocols, or
excluded if it is not needed.

IIOP is the default protocol for TAO. If no action is taken to specifically
declare an alternative protocol, IIOP will be automatically available. This also
means that an endpoint will be selected for any TAO server that does not
otherwise specify its endpoint(s).

15.6.1 Loading and Declaring the Protocol
The IIOP protocol is included in the TAO library. If IIOP is the only protocol
used, no action is required to load and declare the protocol. However, if IIOP
is to be used along with other protocols, then it must be explicitly declared,
along with the other protocols. As mentioned in 15.5, this is done through an
argument to the resource factory. The name of the IIOP protocol factory is
“IIOP_Factory,” as shown in the following example:

    static Resource_Factory "-ORBProtocolFactory IIOP_Factory ... " 
15.6.2 Address Definition

If your TAO server must always be available on a well-known address, you must specify its endpoint(s) as arguments to `CORBA::ORB_init()`.

Often, programs pass command-line arguments to `CORBA::ORB_init()`, so you can also specify endpoints on the command line.

You can also use an implicit endpoint declaration with IIOP-based servers. These endpoints choose a port, possibly from a constrained range, and listen on that same port across multiple network interfaces.

Occasionally, it is desirable to limit the server to a particular network interface, with or without regard to a particular port. Explicit declaration of an endpoint for TAO servers is done with the `-ORBListenEndpoints` option. See 19.8.10 for a full explanation of this option.

When making endpoint specifications, IIOP uses the prefix `iiop://` followed by an optional host name, port number, and any per-endpoint options.

A canonical, or fully-qualified, name should be used whenever supplying a host name to avoid possible inconsistencies in mapping between the host name and the IP address actually used.

Here is a sample IIOP endpoint:

```
server -ORBListenEndpoints iiop://example.ociweb.com:12345
```

Occasionally, IIOP is used in environments where DNS is not available, or is not desirable. In those situations, the IP address of the host is used in place of the host name. For example:

```
server -ORBIGDotDecimalAddresses 1 -ORBListenEndpoints
iiop://192.168.1.10:3000
```

When you set `-ORBIGDotDecimalAddresses 1`, TAO places the IP address in the profile. Otherwise, TAO places whatever is specified on the command line in the profiles.

Communication between processes on the same host may be optimized by using the loopback interface. This is achieved by using the reserved loopback IP address `127.0.0.1`.
According to RFC 1912 and several other RFCs, the name localhost should always resolve to the reserved loopback address 127.0.0.1. Thus, it is permissible to use localhost to identify the host portion of an endpoint. However, some machines do not have name resolution properly configured, so it is safest to always specify 127.0.0.1.

TAO's IIOP pluggable protocol tries to find all IP-based network interfaces. With no endpoint specification, TAO listens on all interfaces on an ephemeral port, and publishes profiles for all the interfaces it can find.

TAO usually gets this right, but finding IP interfaces requires different techniques and APIs on different platforms, so there’s always a possibility TAO might miss a more exotic interface.

You can optionally specify only a port, with profiles published for all interfaces TAO can find:

    server -ORBListenEndpoints iiop://:12345

Or, you can specify just a host name, allowing for port selection from the ephemeral range:

    server -ORBListenEndpoints iiop://example.ociweb.com:

### 15.6.3 IIOP Options

Endpoints may be extended with optional values added to the end of each declaration. In general, these options are supplied in the form “name=value.” Multiple options can be fed by separating them with an ‘&’ e.g. “name=value&name=value”

#### 15.6.3.1 portspan

The portspan option constrains port selection for an IIOP endpoint to a user-specified limited range. To use this feature, a base port value is specified as shown above. The “span,” or number of ports, including the base, is given as the portspan value. For example:

    server -ORBListenEndpoints iiop://example.ociweb.com:12345/portspan=5

This instructs the server to open an endpoint on the specified interface on any port from 12345 and 12349.
The `portspan` option is particularly useful if a server or collection of services is to be accessed through a firewall.

### 15.6.3.2 hostname_in_iior

This option is used to explicitly specify the host name used in IORs. The server ORB will not validate the specified host name. This option allows the same host name to be used in the IORs of objects from multiple servers. This technique can be used for redundancy or as a load-balancing strategy.

For example, two separate servers could be started on separate hosts with the same `hostname_in_iior` option value, as follows:

```bash
# Start a server on host 10.20.1.100:
server -ORBListenEndpoints iiop://10.20.1.100:12345/hostname_in_iior=malory

# Start another copy of the server on host 10.20.1.200:
server -ORBListenEndpoints iiop://10.20.1.200:12345/hostname_in_iior=malory
```

In the above example, the IORs generated by both servers would have the host name `malory` encoded within their IIOP profiles.

### 15.7 IIOP_Lite

IIOP-Lite a TAO-specific variation of the standard IIOP protocol. The difference between this and IIOP is in the format of the GIOP and request message headers. The “lite” form of the headers makes a number of assumptions to save space. For example, applications operating in homogeneous environments do not require all of the standard GIOP and request message header fields. Specifically, the “lite” form of the protocol removes the `magic number`, `version`, and `byte-order-flag` fields from the GIOP header, and the `service context` field from the GIOP request header. The resulting savings of several bytes per request can improve performance. Message transfer overhead and thus latencies are also reduced.

Since the “lite” version of the protocol removes fields from GIOP messages, it can only be used in homogeneous environments (e.g., where all the connected platforms have the same byte order and where TAO is the only ORB). Furthermore, the “lite” version of the protocol must be used for all connected
processes. Results are unpredictable if client and server are not both using the “lite” version of the protocol.

Note
The performance and message transfer overhead savings from using the IIOP_Lite pluggable protocol are small, thus its use is rarely justified.

15.8 UIOP

UIOP stands for the Unix Inter-ORB Protocol. The TAO UIOP specifies how GIOP messages are sent, such that invocations may be made from client to server over Unix Domain Sockets, also known as local IPC. Local IPC often offers better performance for messaging between processes on the same host, while still presenting application developers a sockets-based interface. Though Unix Domain Sockets are supported on many platforms, a few platforms, such as Win32, pSOS and QNX, do not support them.

Unlike IIOP, UIOP is not available by default. When available, servers will present UIOP profiles only when you explicitly specify a UIOP endpoint.

15.8.1 Loading and Declaring the Protocol

The UIOP factory and its components are included in the TAO_Strategies library. It is possible to directly link the strategies library to your application by adding TAO_Strategies to the list of libraries in your makefile or project settings. UIOP, along with the other protocols included in the strategies library, may be made available by default as well by including the following TAO-specific header in your application’s main source file:

```c
#include <tao/Strategies/advanced_resource.h>
```

Including this file makes all of the protocols supplied in the strategies library available to the application.

You may want to limit the available protocols, which you can do by specifying the UIOP protocol factory name, UIOP_Factory, as an argument to the -ORBProtocolFactory option when initializing the resource factory. For example:

```c
static Advanced_Resource_Factory "-ORBProtocolFactory UIOP_Factory"
```
Either the default Resource Factory or the Advanced Resource Factory may be used when the strategies library is statically linked.

If you also want to use IIOP, you will need to include IIOP_Factory as an additional argument to -ORBProtocolFactory.

You can also load the UIOP protocol dynamically through the service configurator. This removes the need to include the TAO-specific header or explicitly link the strategies library. However, the TAO_Strategies library must exist in your platform’s library search path (e.g., LD_LIBRARY_PATH on many Unix systems). You also must use the following dynamic directives in your svc.conf file (see 18.3.2 for more information):

```c

dynamic UIOP_Factory Service_Object*
    TAO_Strategies: make_TAO_UIOP_Protocol_Factory () ""

dynamic Advanced_Resource_Factory Service_Object*
    TAO_Strategies: make_TAO_Advanced_Resource_Factory ()
        "-ORBProtocolFactory UIOP_Factory"
```

Once the UIOP protocol factory is loaded, the ORB will be able to communicate via this protocol.

### 15.8.2 Address Definition

For TAO servers to generate UIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the -ORBListenEndpoints option. See 19.8.10 for more general information on this option.

UIOP Endpoints are composed of the prefix uiop:// followed by an optional Unix domain rendezvous point. Unix domain rendezvous points appear as directory entries in the local file system. As such, they are represented as a file path, such as /tmp/my_uiop_endpoint. To use this rendezvous point as a UIOP endpoint, the declaration is:

```bash
-ORBListenEndpoints uiop:///tmp/my_uiop_endpoint
```

You can cause TAO to create an ephemeral rendezvous point (similar to ephemeral ports in TCP) by using an empty endpoint declaration. You still must explicitly declare the endpoint:

```bash
-ORBListenEndpoints uiop://
```
Using Pluggable Protocols

TAO generates UIOP rendezvous points using a facility like the system function `tempnam()` or `tmpnam()`, depending on your platform. It places rendezvous points in the temporary directory defined for the environment, usually `/tmp`, and generates a unique name prefixed with “TAO”. These ephemeral rendezvous points are removed upon normal shutdown of the ORB.

15.8.3 Guidelines

The rendezvous point for UIOP is any valid path and filename that the ORB has permission to read from and write to. However, UIOP rendezvous points have the same restrictions as local IPC. The following guidelines will help you use TAO’s UIOP pluggable transport protocol successfully:

- **Limit the length of the endpoint.** For the greatest platform portability of your code, ensure that rendezvous points are no longer than 99 characters, including path and filename. This limit may be greater on specific platforms, but the POSIX.1g standard dictates a limit of no more than 100 characters, including null terminator. Endpoints longer than what the platform supports are truncated, and a warning is issued.

- **Use absolute paths whenever possible.** This avoids the trouble of ensuring that the client and server are started in a particular directory. For instance, declaring a relative endpoint, such as `-ORBListenEndpoints uiop://myuiop`, causes a local IPC rendezvous point called `myuiop` to be created in the current working directory. If the client is not started in the same directory, it cannot locate the rendezvous point, and fails to communicate with the server. On the other hand, using an absolute path insures that the client knows exactly where to find the rendezvous point. Relative paths can also expose one to security risks through man-in-the-middle attacks.

- **Insure accessibility.** It is up to the user to make sure that a given UIOP rendezvous point is accessible by both the server and the client.

15.9 SHMIOP

SHMIOP stands for the Shared Memory Inter-ORB Protocol. The TAO SHMIOP specifies the proper encoding of IORs, and how GIOP messages are
sent, such that invocations may be made from client to server via shared memory.

TAO’s SHMIOP implementation uses ACE memory-mapped files as the shared-memory substrate. It does not use the System V Unix shared-memory IPC mechanism.

Like UIOP, SHMIOP is useful only between processes that can share memory, typically limited to same-host processes. This protocol is only available on platforms that support shared memory. Though supported on many platforms, it is unavailable on a few, such as some single process real-time operating systems. This protocol can provide a performance advantage over IIOP, and is available on some platforms that do not support UIOP, such as Win32.

An interesting feature of TAO’s SHMIOP protocol is how it solves the notification problem. A requirement of pluggable protocols is to generate I/O events. This is solved in SHMIOP by the use of a TCP/IP connection through the loopback interface, ensuring that the ORB receives timely notification of data availability.

### 15.9.1 Loading the Protocol

The SHMIOP factory and its components are in the TAO Strategies library. To directly link the strategies library to your application, add TAO_Strategies to the list of libraries in your makefile or project settings. SHMIOP, along with the other protocols included in the strategies library, may be made available by default by including the following TAO-specific header in your application’s main source file:

```
#include <tao/Strategies/advanced_resource.h>
```

By including this file, you make all of the protocols supplied in the strategies library available to your application.

To limit the available protocols, specify the SHMIOP protocol factory name, SHMIOP_Factory, as an argument to the -ORBProtocolFactory option when initializing the resource factory. For example:

```
static Advanced_Resource_Factory "-ORBProtocolFactory SHMIOP_Factory"
```
Either the default Resource Factory or the Advanced Resource Factory may be used when the strategies library is statically linked.

If you also want to use IIOP, then include IIOP_Factory as an additional argument to -ORBProtocolFactory.

You can also load the SHMIOP protocol dynamically through the service configurator, which makes it unnecessary to include the TAO-specific header or explicitly link the strategies library. However, the TAO_Strategies library must exist in your platform’s library search path (e.g., LD_LIRBARY_PATH on many Unix systems). You also must use the following dynamic directives in your svc.conf file (see 18.3.2 for more information):

```
dynamic SHMIOP_Factory Service_Object*
    TAO_Strategies::make_TAO_SHMIOP_Protocol_Factory () ""

dynamic Advanced_Resource_Factory Service_Object*
    TAO_Strategies::make_TAO_Advanced_Resource_Factory ()
    "-ORBProtocolFactory SHMIOP_Factory"
```

### 15.9.2 Configuring the SHMIOP Factory

The SHMIOP factory takes two options that globally affect the transports created by it. These options control the name and the size of the memory-mapped file that is used for virtual-memory backing storage.

- `-MMAPFilePrefix prefix`. This option sets the prefix on the backing store filename. This prefix may be any string that is valid as the start of a filename. If this option is omitted, then “MEM_Acceptor_” is used by default. The remainder of the filename, appended to the prefix, is the decimal representation of the port number used for I/O event notification.

- `-MMAPFileSize size`. This option sets the size of the backing store file. The size value is in bytes. The default size is 10k bytes.

The largest message size that can be handled by SHMIOP is the size of the backing store. The minimum size of the map file should be at least the size of the message, including message headers and other overhead. You should reserve a file size equal to the expected maximum size of a request plus the expected maximum size of a reply, times a safety factor:

\[
(MAX\_REQUEST\_SIZE + MAX\_REPLY\_SIZE) \times SAFETY\_FACTOR
\]
For example, if you expect the maximum request size in your application to be 4k bytes, the maximum reply size to be 16k bytes, and you assume a safety factor of two, you would have:

\[(4k + 16k) \times 2 = 40k\]

You specify these as options to the SHMIOP factory in the `svc.conf` file. For example:

```
static SHMIOP_Factory "-MMAPFilePrefix server_shmiop -MMAPFileSize 40960"
```

### 15.9.3 Address Definition

For TAO servers to generate SHMIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the `-ORBListenEndpoints` option. See 19.8.10 for more information on this option.

SHMIOP endpoints are composed of the prefix `shmiop://` followed by an optional port number. For example:

```
-ORBListenEndpoints shmiop://12345
```

The port number specifies the TCP port used for notification of new data. TAO uses an ephemeral port if you do not specify a port.

Although SHMIOP endpoints do not accept a host name in the declaration, the canonical name of the host appears in the profile, which is embedded in object references. This name is used to prevent processes on other hosts from attempting to use that SHMIOP profile.

### 15.10 DIOP

DIOP stands for Datagram Inter-ORB Protocol and is a UDP-based transport protocol. In TAO 1.4a, this protocol is only partially implemented and as such, has some restrictions that will be covered later in this section. The DIOP implementation uses connectionless UDP sockets, and therefore is intended for use as a lower-overhead protocol for certain classes of applications.
The original motivation for this protocol came from applications that used only oneway operations.

### 15.10.1 Loading the Protocol

The DIOP factory and its components are included in the TAO_Strategies library. To directly link the strategies library to your application, add TAO_Strategies to the list of libraries in your makefile or project settings. DIOP, along with the other protocols included in the strategies library, may be made available by default by including the following TAO-specific header in your application’s main source file:

```c
#include <tao/Strategies/advanced_resource.h>
```

By including this file, you make all of the protocols supplied in the strategies library available to your application.

To limit the available protocols, specify the DIOP protocol factory name, DIOP_Factory, as an argument to the -ORBProtocolFactory option when initializing the resource factory. For example:

```c
static Advanced_Resource_Factory "-ORBProtocolFactory DIOP_Factory"
```

Either the default Resource Factory or the Advanced Resource Factory may be used when the strategies library is statically linked.

You can also load the DIOP protocol dynamically through the service configurator, making it unnecessary to include the TAO-specific header, or explicitly link the strategies library. However, the TAO_Strategies library must exist in your platform’s library search path (e.g., LD_LIBRARY_PATH on many Unix systems). You also must use the following dynamic directives in your svc.conf file (see 18.3.2 for more information):

```c
dynamic DIOP_Factory Service_Object* 
    TAO_Strategies: make_TAO_DIOP_Protocol_Factory () ""

dynamic Advanced_Resource_Factory Service_Object* 
    TAO_Strategies: make_TAO_Advanced_Resource_Factory () 
    "-ORBProtocolFactory DIOP_Factory"
```

Once you have the DIOP protocol factory loaded, the ORB will be able to communicate via this protocol.
15.10.2 Address Definition

For TAO servers to generate DIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the `-ORBListenEndpoints` option. See 19.8.10 for more information on this option.

Endpoints for DIOP are composed of the prefix `diop://` followed by a host and port combination, similar to IIOP endpoints. An example of a DIOP endpoint is:

```
-ORBListenEndpoints diop://example.ociweb.com:12345
```

15.10.3 Notes and Restrictions

Keep the following points in mind when using the DIOP protocol:

- There are no connections, and therefore no state; requests from different clients all arrive at the same socket!
- The thread-per-connection concurrency model is supported.
- Only `oneway` operation invocations are supported.

DIOP was developed for applications satisfying the following assumptions:

- UDP communications are nearly 100 percent reliable (e.g., IP over ATM). Even under less reliable conditions, DIOP can be used by restricting the application to use `oneway` operations for both requests and responses in combination with application-level time outs.
- TCP is inappropriate, due to its sluggishness on sudden disconnections (e.g., it must be possible to “hot swap” CPU cards without impacting an ORB’s communications with a particular CPU). For this reason, no state is kept on the client side of the DIOP protocol.
- No GIOP message (and therefore no IDL signature) has a size greater than `ACE_MAX_DGRAM_SIZE` (4 kilobytes). Thus, no data sent via DIOP can be larger than 4 kilobytes.

Note that support for fragmented messages, as specified in GIOP 1.2, may alleviate this restriction, and may be added to a future DIOP implementation.
15.11 **SSLIOP**

SSLIOP stands for the Secure Sockets Layer (SSL) Inter-ORB Protocol. This protocol is defined by the OMG as part of the CORBA Security Service specification. SSLIOP uses GIOP as a messaging protocol and SSL as the transport protocol. It is a drop-in replacement for IIOP, providing secure communication between hosts.

TAO's SSLIOP pluggable protocol implementation supports both the standard IIOP transport protocol and the secure IIOP over SSL transport protocol. No changes were made to the core TAO source code to provide SSL support, nor does TAO contain any security-related hooks. Details about TAO’s SSLIOP and other security-related issues are covered in Chapter 29, particularly 29.3.

15.11.1 **Loading the Protocol**

The SSLIOP factory and its components are included in a separate TAO_SSLIOP library. See 29.5 for details on building and using this library. Specifically, 29.6.2 covers the various initialization options that are accepted by the SSLIOP Factory.

15.11.2 **Address Definition**

SSLIOP endpoints are nearly identical to IIOP endpoints. The same prefix, `iiop://` is used, along with host name and port number. The only difference is that SSLIOP takes a separate per-endpoint option, `ssl_port`, that allows explicit specification of both the secure and insecure endpoints. The `portspan` option of the IIOP connector is not used by SSLIOP. Here is an example of an endpoint supplying both an insecure and a secure port:

```
-ORBListenEndpoints iiop://example.ociweb.com:12000/ssl_port=12001
```

15.11.3 **Portspan**

The `portspan` option, which was described earlier in section 14.6.3.1 as an IIOP option, is also available for the SSLIOP endpoint. When used in conjunction with `ssl_port` as shown below:

```
-ORBListenEndpoints
iiop://example.ociweb.com:12000/ssl_port=12001&portspan=5
```
15.12 MIOP/UIPMC

MIOP stands for Multicast Inter-ORB Protocol and UIPMC stands for Unreliable IP Multicast. UIPMC underlies the OMG’s Multicast Inter-ORB Protocol, defined in the MIOP specification. (OMG Document ptc/01-11-08 is the Final Adopted Specification.) The current MIOP version is 1.0. The MIOP protocol wraps a GIOP request in a MIOP Packet and transmits it via an underlying connectionless transport protocol (e.g., UDP/IP). An MIOP Packet is defined as the MIOP Packet Header information, which represents the state information for a single MIOP Packet, as well as the raw GIOP data (body) contained in the rest of the MIOP Packet. An MIOP Packet will be sent and later reassembled on the receiving side. MIOP Packets are the atomic pieces that comprise a Packet Collection. A Packet Collection is comprised of one or more MIOP Packets and is defined as a complete, packaged, GIOP request message or request message fragment. Only GIOP request messages and associated request message fragments are allowed in an MIOP Packet in a Packet Collection.

**Note**

MIOP packet reassembly is not yet supported in TAO. Thus, the maximum request size that can be transmitted via MIOP/UIPMC is currently limited to roughly 5-6k bytes depending upon the platform.

TAO’s MIOP support enables servants to receive requests sent to multicast addresses. A GroupId that identifies the multicast group is created and associated with one or more servants. The UIPMC pluggable protocol is used to send and receive multicast requests. MIOP/UIPMC is currently usable only for oneway operations. Servers (receivers) join an IP Multicast group and listen on the same group id for requests.

See $TAO_ROOT/orbsvcs/tests/Miop/McastHello for an example that uses MIOP/UIPMC.
15.12.1 Loading the Protocol

The UIPMC factory and its components are included in the TAO_PortableGroup library. You can load the UIPMC protocol dynamically through the service configurator. The TAO_PortableGroup library must exist in your platform’s library search path (e.g., LD_LIBRARY_PATH on many Unix systems). You also must use the following dynamic directives in your svc.conf file (see 18.3.2 for more information):

```c
dynamic UIPMC_Factory Service_Object*
TAO_PortableGroup:_make_TAO_UIPMC_Protocol_Factory() ""

static Resource_Factory "-ORBProtocolFactory IIOP_Factory -ORBProtocolFactory UIPMC_Factory"
```

Once you have the UIPMC protocol factory loaded, the ORB will be able to communicate via this protocol. The UIPMC protocol factory must be loaded and configured on both the client and server sides. In addition, the server side must add the following directive to load the Portable Group Loader service object.

```c
dynamic PortableGroup_Loader Service_Object*
TAO_PortableGroup:_make_TAO_PortableGroup_Loader() ""
```

15.12.2 Address Definition

The MIOP specification provides the following corbaloc Object URL syntax for specifying an MIOP profile:

```xml
<corbaloc> = "corbaloc:"<obj_addr_list>{""<key_string>}
<obj_addr_list> = [<obj_addr> "," ]* <obj_addr>
<obj_addr> = <prot_addr>
<prot_addr> = <iiop_prot_addr> | <miop_prot_addr>
<miop_prot_addr> = <miop_prot_token><miop_addr>
<miop_prot_token> = "miop"
<iiop_prot_token> = "iiop"
<miop_addr> = <version><group_addr>[;<group_iiop>]
<version> = <major> "." <minor> "@" | empty_string
<group_addr> = <group_id>"/"<ip_multicast_addr>
<group_iiop> = <iiop_prot_token>":"<version> <hostname>"/"<port> "/" <objecy_key>
<ip_multicast_addr> = <classD_IP_address> | <IPv6_address> "::" <port>
<classD_IP_address> = "224.0.0.0" - "239.255.255.255"
<port> = number (default to be defined)
<group_id> = <group component version>"-"<group_domain_id>"-"
```
An example of a valid MIOP corbaloc Object URL (taken from the example in $TAO_ROOT/orbsvcs/tests/Miop/McastHello) is:

corbaloc:miop:1.0@1.0-TestDomain-1/225.1.1.225:1234;

Here, the group id is 1.0-TestDomain-1 and the IP multicast address is 225.1.1.225 at port 1234. (Any valid multicast address will do.) Multiple servers using the MIOP/UIPMC protocol can share this same group id and address; all the servers will be able to receive client requests directed to that group.

The -ORBListenEndpoints option is not used to specify MIOP profiles as with TAO’s other pluggable protocols. Instead, servers should generate an object reference from the group’s corbaloc Object URL (using CORBA::ORB::string_to_object()), then use the POA’s new create_id_for_reference() operation to generate a PortableServer::ObjectId from a group’s object reference. This object id is then associated with servants via the POA’s usual activate_object_with_id() or similar operation. The following server code fragment shows how the above steps can be accomplished:

```
// ORB and POA initialization code not shown.
// Set up the object group’s corbaloc Object URL.
CORBA::String_var group_url =
   CORBA::string_dup ("corbaloc:miop:1.0@1.0-TestDomain-1/225.1.1.225:1234");

// Generate a group object reference.
CORBA::Object_var group_reference = orb->string_to_object (group_url.in());

// Create an Object Id for the group reference.
PortableServer::ObjectId_var id =
   poa->create_id_for_reference (group_reference.in ()

// Create and activate a servant.
Messenger_i servant;
poa->activate_object_with_id (id.in(), &servant);
```
// Activate the POAManager and accept client requests as usual.

Clients can simply convert the group’s corbaloc Object URL to an object reference using CORBA::ORB::string_to_object(), then invoke oneway operations on the object reference as usual. Since only oneway operations are supported, clients cannot use a normal _narrow() function to narrow the object reference since doing so may cause a two-way _is_a() operation to be invoked; instead, clients should use _unchecked_narrow(). The following client code fragment shows the steps we have just described:

// ORB initialization code not shown.
// Set up the object group’s corbaloc Object URL.
CORBA::String_var group_url =
    CORBA::string_dup("corbaloc:miop:1.0@1.0-TestDomain-1/225.1.1.225:1234");

// Generate a group object reference.
CORBA::Object_var group_reference = orb->string_to_object (group_url.in());

// Narrow the object reference (use _unchecked_narrow() since _narrow() may // result in a two-way _is_a() operation invocation).
Messenger_var messenger = Messenger::_unchecked_narrow (group_reference.in ());
if (CORBA::is_nil (messenger.in())) {
    std::cerr << "Could not narrow group reference." << std::endl;
    ACE_OS::exit(-1);
}

// Invoke oneway operations on the group reference.

15.12.3 Notes and Restrictions
Keep the following points in mind when using MIOP:

• There are no connections, and therefore no state; requests from different clients are all sent to the same multicast address and will be received by all servers that have joined the multicast group!

• The thread-per-connection concurrency model is supported.

• Only oneway operation invocations are supported.
• IP multicast communications are unreliable, therefore MIOP/UIPMC should not be used where reliable transmission of requests/data is required.

• Since TAO’s support of MIOP does not yet support message fragmentation and MIOP Packet reassembly, request sizes are limited to 5-6 kilobytes, depending upon the platform.

15.13 HTIOP

HTIOP stands for HTTP Tunneling Inter-ORB Protocol. It is a TAO-specific protocol that allows inter-ORB communications to be transmitted across a firewall. It does this by layering GIOP messages over HTTP packets.

HTIOP support in TAO consists of two libraries:

• The ACE_HTBP library, found in $ACE_ROOT/protocols/ace/HTBP, implements the low-level HTBP protocol. This protocol allows for communications to take place through a firewall between “inside” and “outside” peers. HTBP supports both “direct” and “indirect” connections through the firewall. The default is indirect, which requires the use of an HTTP proxy. HTBP can be used independently of TAO.

• The TAO_HTIOP library, found in $TAO_ROOT/orbsvcs/orbsvcs/HTIOP, implements HTIOP on top of HTBP.

HTIOP is an asymmetric protocol. This means that applications inside the firewall must be configured differently than applications outside the firewall. Inside peers are the only ones that may initiate connections.

Note If a peer-to-peer relationship is desired wherein CORBA invocations flow in either direction, then bi-directional GIOP must be used. See 7.4 for more information on bi-directional GIOP.

One of the challenges imposed by the use of an HTTP proxy is that, if a connection is idle for long enough, it may be closed. This means the inside peer must establish a new connection by sending another request. Since the outside peer cannot open a connection, it must queue messages destined to the
inside peer until the inside peer opens a new connection. Once a connection has been established, all the queued messages are sent.

A second challenge is that, while a proxy will open multiple TCP/IP connections to a server (the outside peer), when faced with multiple simultaneous HTTP requests, it will reuse those connections at will to forward any subsequent requests. This means that a socket is associated with a particular HTIOP session for only one HTTP request/reply cycle.

Apart from loading the protocol and declaring an endpoint, no modifications to application code are required to use HTIOP.

HTBP isolates HTIOP from differences in HTTP proxies by defining a filter class that encapsulates the particular characteristics of the proxy that is being used and that is responsible for marshaling and demarshaling binary data. At the time of this writing, HTBP has a single filter class that works with the Squid HTTP proxy. More information on the Squid web proxy cache can be found via <http://www.theaceorb.com/references/>. This filter may also be used as a null filter when the inside and outside peers are directly connected (i.e., a proxy is not being used), which is mainly useful during testing.

15.13.1 Loading the Protocol

HTIOP is loaded via the service configurator. Since HTIOP is an asymmetric protocol, the configurations are different for the server and client sides. Since the server resides outside the firewall, its configuration is quite straightforward. Here is an example of a server-side configuration:

```c
dynamic HTIOP_Factory Service_Object *
    TAO_HTIOP::_make_TAO_HTIOP_Protocol_Factory () ""

static Resource_Factory "-ORBProtocolFactory HTIOP_Factory"
```

The HTIOP_Factory protocol factory accepts the following initialization parameters:

- `-config <filename>`—Specifies the name of a text file that contains an HTBP-specific configuration. See Table 15-2 for a list of possible HTBP configuration parameters. The `-config` parameter is optional.

- `-env_persist <filename>`—Specifies the name of a memory mapped file that contains a previously saved HTBP configuration, or to which a new HTBP configuration will be saved. If both `-config` and `-env_persist` are specified, the memory mapped file will be loaded.
first, then the text file will be interpreted. The new configuration will then be persisted in the memory mapped file.

- `-win32_reg`—If this parameter is provided, the HTBP configuration will be saved to the Windows registry (in the HTBP section) rather than to a memory mapped file. This feature is available only on Windows.

- `-inside (-1|0|1)`—Explicitly declare whether the peer’s endpoint is inside or outside the firewall. The default setting of `-1` causes the `HTIOP.Factory` to use the `ACE_HTBP_Environment`'s `proxy_host` setting to determine if it the peer is inside or outside the firewall. An explicit setting of `1` means the peer’s endpoint is inside the firewall; an explicit setting of `0` means it is outside the firewall.

You only need to explicitly specify if the peer is inside or outside the firewall when you are testing without a real HTTP proxy. As stated previously, HTIOP is an asymmetric protocol: only peers inside the firewall can initiate connections; peers outside the firewall cannot initiate connections to inside peers.

HTBP is configured independently of the HTIOP protocol factory. The HTBP configuration contains a single `[htbp]` section that contains a list of name/value pairs to configure HTBP. Table 15-2 lists the configuration parameters HTBP accepts:

**Table 15-2 HTBP Configuration Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proxy_host</td>
<td>The host name of the HTTP proxy through which all requests will be sent.</td>
</tr>
<tr>
<td>proxy_port</td>
<td>The port number on which the HTTP proxy is listening and through which all requests will be sent.</td>
</tr>
</tbody>
</table>

In the following example, we show the configuration of an HTIOP client in which a proxy is not being used. Note that the client must explicitly specify that it is an inside peer.

```c
    dynamic HTIOP.Factory Service_Object *
        TAO_HTIOP:_make_TAO_HTIOP_Protocol_Factory () "-inside 1"
    static Resource.Factory "-ORBProtocolFactory HTIOP.Factory"
```

In this example, we show a client configuration that is using a proxy. The HTBP configuration parameters are specified in a separate configuration file called `HTBP_Config.txt`. 
**Using Pluggable Protocols**

```c
  dynamic HTIOP_Factory Service_Object *
  TAO_HTIOP_make_TAO_HTIOP_Protocol_Factory () "-config HTBP_Config.txt"
static Resource_Factory "-ORBProtocolFactory HTIOP_Factory"
```

Here we show possible contents of the HTBP_Config.txt file.

```
[htbp]
proxy_port=3128
proxy_host=proxy.acme.com
```

### 15.13.2 Address Definition

For TAO servers to generate HTIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the `-ORBListenEndpoints` option. See 19.8.17 for more information on this option.

HTIOP endpoints are composed of the prefix `htiop://`, followed by the host name and port number. For example:

```bash
-ORBListenEndpoints htiop://malory.acme.com:12345
```

### 15.13.3 HTIOP Options

#### 15.13.3.1 hostname_in_ior

This option is used to explicitly specify the host name used in IORs. For example:

```bash
```

See 15.6.3.2 for more information on the `hostname_in_ior` option.

### 15.14 SCIOP

SCIOP stands for SCTP Inter-ORB Protocol. It layers the TAO pluggable protocol framework atop the Stream Control Transmission Protocol (SCTP).

SCTP is defined in Internet Engineering Task Force (IETF) Requests for Comments (RFC) 2960 as the transport layer to carry signaling messages over
IP networks. According to the RFC, “SCTP is a reliable transport protocol operating on top of a connectionless packet network such as IP. It offers the following services to its users:

- acknowledged error-free non-duplicated transfer of user data,
- data fragmentation to conform to discovered path MTU size,
- sequenced delivery of user messages within multiple streams, with an option for order-of-arrival delivery of individual user messages,
- optional bundling of multiple user messages into a single SCTP packet, and
- network-level fault tolerance through supporting of multi-homing at either or both ends of an association.

The design of SCTP includes appropriate congestion avoidance behavior and resistance to flooding and masquerade attacks.”

TAO’s implementation of SCIOP depends upon an external implementation of the SCTP APIs. Currently, the only supported SCTP implementation is the Linux Kernel Stream Control Transmission Protocol (lksctp). lksctp is available in the Linux kernel source tree in versions 2.5.36 and later. For this reason, TAO’s SCIOP support is currently limited to the Linux platform.

15.14.1 SCTP Association

SCTP provides a means for each SCTP endpoint to provide the other endpoint (during association startup) with a list of transport addresses (i.e., multiple IP addresses in combination with an SCTP port) through which that endpoint can be reached and from which it will originate SCTP packets. The association spans transfers over all of the possible source/destination combinations that can be generated from each endpoint’s lists. Each GIOP connection is mapped to a single SCTP association. At startup, a primary path is selected, which is indicated in the IOR profile. This primary path is used for normal communication. Using SCIOP, a TAO application is able to access the fault tolerance capabilities of the SCTP protocol.

15.14.2 Loading the Protocol

The SCIOP implementation is included in the TAO_Strategies library. It can be loaded and configured either dynamically or statically.
SCIOP can be dynamically loaded via the service configurator. An example configuration file is shown below:

```plaintext
dynamic SCIOP_Factory Service_Object*
   TAO_Strategies: make_TAO_SCIOP_Protocol Factory () ""
dynamic Advanced_Resource_Factory Service_Object*
   TAO_Strategies: make_TAO_Advanced_Resource_Factory ()
   "-ORBProtocolFactory SCIOP_Factory"
```

See 15.8.1 for information on statically loading the `TAO_Strategies` library.

### 15.14.3 Address Definition

SCIOP addresses must encapsulate the SCTP association information. Multiple host names separated by a `+` are allowed for a single endpoint. As with TAO’s other pluggable protocols, the `-ORBListenEndpoints` option is used to create an SCIOP endpoint. SCIOP endpoints are composed of the prefix `sciop://`, followed by one or more host names and a port number. Some sample SCIOP endpoint specifications are shown below:

```plaintext
-ORBListenEndpoints sciop://malory:12345
-ORBListenEndpoints sciop://malory+arthur+gawain:12345
```

### 15.14.3.1 hostname_in_iors

This option is used to explicitly specify the host name used in IORs. For example, the following option:

```plaintext
-ORBListenEndpoints sciop://malory+arthur+gawain:12345/hostname_in_iors=host4
```

will cause the ORB to encode `host4` in the SCIOP profiles of the IORs that it generates (i.e., `host4` will be the primary host for the service).

See 15.6.3.2 for more information on the `hostname_in_iors` option.

### 15.15 Combining Protocols

You may use multiple protocols simultaneously with TAO. Using the `-ORBListenEndpoints` option, you can instruct the ORB to listen on multiple endpoints for different protocols. For example, to instruct the IIOP
protocol to listen on an ephemeral port on 192.168.1.6, and the SHMIOP protocol to listen on port 12345, use the following command:

    -ORBListenEndpoints iiop://192.168.1.6 -ORBListenEndpoints shmiop://12345

or alternatively:

    -ORBListenEndpoints iiop://192.168.1.6;shmiop://12345

Creating multi-protocol endpoints, as in the example above, requires the service configurator to load and initialize the protocol-specific factory objects. The IIOP factory is initialized by default unless another protocol is explicitly loaded via the service configurator. Therefore, the example above requires both IIOP and SHMIOP factories to be statically or dynamically initialized. The dynamic directive for this is shown below:

dynamic IIOP_Factory Service_Object*
    TAO_Strategies: make_TAO_IIOFFact]() ""

dynamic SHMIOP_Factory Service_Object*
    TAO_Strategies: make_TAO_SHMIOP_Fact]() ""

dynamic Advanced_Resource_Factory Service_Object*
    TAO_Strategies: make_TAO_Advanced_Resource_Factory()
    "-ORBProtocolFactory IIOP_Factory -ORBProtocolFactory SHMIOP_Factory"

When a multi-endpoint server generates an IOR, that IOR will contain a profile for each protocol in the order in which they appear on the command line. Without using the capabilities of real-time CORBA, there is no standard way for the client to specify at run-time which protocol to use for a particular invocation. A TAO server and a TAO client, however, can leverage the fact that the client connection code iterates over profiles in the IOR in the order in which they appear. Thus, for the endpoints specified above, the client ORB first tries to establish a connection and send a request using IIOP and, if that fails, tries SHMIOP.

Reversing the order of the endpoints on the server’s command line causes the client ORB to first attempt to use SHMIOP, then IIOP:

    -ORBListenEndpoints shmiop://12345 -ORBListenEndpoints iiop://192.168.1.6
The standard and more portable way to accomplish this is through the use of the real-time-CORBA protocol selection policies. An example of this can be found in $TAO_ROOT/tests/RTCORBA/Client_Protocol and in 9.3.12.
CHAPTER 16

Developing Pluggable Protocols

16.1 Introduction

The previous chapter discussed the use of the different protocols that are part of TAO. This chapter covers the internal framework used to implement these protocols, with a focus on how to develop your own protocols for use with TAO.

The OMG is standardizing such a framework via the Extensible Transport Framework (ETF) specification. The Final Adopted Specification (OMG Document ptc/04-03-03) document is in the Version 1.0 finalization process. However, because TAO’s pluggable protocols framework predates the OMG’s ETF specification by several years, all APIs discussed in this section are TAO specific. There is currently no plan to support the ETF within TAO.

As with the use of the existing TAO protocols, the main motivator for using custom protocols is typically higher performance, although reduced bandwidth consumption or security may also motivate their use. If the requirements of your application are such that IIOP or one of the other protocols provided with TAO is not sufficient, then you can take full
Developing Pluggable Protocols

advantage of TAO’s open pluggable-protocol framework to architect your own transport for seamless use with the rest of the TAO ORB.

Note What has been historically referred to as a “pluggable protocol” in TAO is really a pluggable transport. TAO does not currently support pluggable messaging (i.e., only GIOP is supported), nor is it possible to choose alternative marshaling schemes (i.e., only CDR is supported). However, for historical reasons, pluggable transports in TAO are referred to as pluggable protocols.

Generally, as a CORBA application programmer, you need not be concerned with the internal workings of the ORB, beyond the configuration options provided to you by the ORB vendor. When developing pluggable protocols however, you need to be familiar with the patterns, mechanisms, and APIs used by the ORB internal code, so that you can quickly and effectively add your own transport. After covering the requirements that any transport must satisfy to be used for GIOP messaging, this chapter covers the patterns, mechanisms, and APIs used with pluggable protocols, and outlines the exact steps you need to take to develop your custom protocol and integrate it with TAO.

More specifically, we dissect a simple CORBA example program to allow you to gain an overview of the collaborations between the pluggable protocol framework components on both the client side and server side, within the context of an actual program. Then we discuss the detailed responsibilities and interactions of each of these framework components, and lay out the exact code that you must provide with the components for your custom protocol. This chapter illustrates both the static and dynamic relationships between the classes you provide and the generic framework classes that already exist in the TAO internals.

16.2 Pluggable Protocol Requirements

This section discusses the preconditions that must be met by any candidate transport protocol considered for use with TAO.
16.2.1 Transport Behavior
Any transport protocol used in inter-ORB communication must be both connection-oriented and reliable. Basic TCP/IP is a representative example, because it fulfills both of these requirements. UDP is an example of a protocol that does not fulfill these minimum requirements, being neither connection-oriented nor reliable. Of course, a connection-oriented, reliable layer could be built on top of UDP.

Additionally, the candidate protocol must have a way to multiplex I/O events, using either the Unix `select()` system call or its Win32 equivalent, `WaitForMultipleObjects()`. Such event notification may be supplied through an alternate channel if the native transport mechanism does not support it. For example, the shared-memory transport protocol, SHMIOP, uses a socket on the loopback interface to notify the peer connection that data is available for reading. As an extreme example, it may be possible to use some other stimulus, such as signals or semaphores, for notification, but integrating such a mechanism with TAO is difficult and beyond the scope of this book.

16.2.2 Identifier Tags
Transport protocols are identified with a magic number, which is defined by the OMG. Throughout the following class descriptions, all of the constructors defined by the TAO pluggable-protocol framework take a `CORBA::ULong` value as an argument. This value represents the protocol’s tag, which is incorporated into the profile section of an IOR, and therefore must be unique across all CORBA vendors. To obtain a tag assignment when constructing a new protocol, it is necessary to register the protocol with the OMG. Any value may be used as the tag, but failure to register it leaves your code liable to inconsistent behavior when interacting with clients or servers produced by others.

16.3 Overview of Pluggable Protocols in TAO
The pluggable-protocol framework is the mechanism used by the ORB internals for all protocols to establish connections and transmit data between a client and a server. As such, producing specialized protocols for this framework requires that you be familiar with how TAO uses protocols in general. Before tackling the study of TAO’s internal handling of protocols, it
is vital that you have an understanding of certain concurrent and networking
design patterns. In particular, you need to be very familiar with the Acceptor,
Connector, Reactor and Service-Configurator (Component-Configurator)
design patterns, which are documented in *Pattern-Oriented Software
Architecture: Patterns for Concurrent and Networked Objects* (POSA2)

You also need to be familiar with certain general-purpose design patterns,
including the Abstract Factory, Strategy, and Template Method patterns, all
described in *Design Patterns: Elements of Reusable Object-Oriented Software
(GoF) (1995) by Erich Gamma, Richard Helm, Ralph Johnson, and John
Vlissides, as well as the Manager pattern, described in *Pattern Languages Of
Program Design, Volume 3*. Your job of implementing a custom protocol will
be easier if you are familiar with the ACE wrapper facades, patterns, and
frameworks, described in *C++ Network Programming* books by Doug
Schmidt and Steve Huston: *Volume 1: Mastering Complexity Using ACE and
Patterns (C++NPv1) and Volume 2: Systematic Reuse with ACE and
Frameworks (C++NPv2)*. Understanding these patterns and their dynamics
will assist you greatly in understanding the interaction between the TAO
internal code and any pluggable protocol that you wish to integrate with TAO.

One great feature of the TAO pluggable-protocol framework is that no
modifications are necessary to the TAO source code when integrating your
custom protocol. Your protocol code can be defined and implemented in such
a way that it is seamlessly created, linked, loaded, invoked called by the TAO
internal code.

The best way to get an overview of how the pluggable-protocol framework is
designed and how it works in action is to examine a simple CORBA program
to see what happens in the pluggable-protocol framework at various points
during the execution of both the client side and server side. To this end, we
analyze the interactions between a simple example CORBA program and the
TAO pluggable-protocol framework. We examine how both interact with each
other, with the TAO internal code, with the command line arguments, and
with the service-configurator file. After examining the simple CORBA
example, we take up the framework components in detail, so that you can see
exactly what is required of the components you will be developing for your
custom protocol.

The following section contains a high-level view of the ORB code, designed
to give you an orientation of the structure and dynamics of the TAO
16.3 Overview of Pluggable Protocols in TAO

pluggable-protocol framework and the context in which it operates. Later sections provide the details needed for implementing your custom protocol components. The protocol we examine in this example is IIOP. Keep in mind that IIOP is implemented as a pluggable protocol in TAO and thus its behavior mimics, to a very large degree, that of any pluggable protocol used with TAO.

16.3.1 A Simple Example: Server Side

This section presents the code for a simple CORBA server and discusses what happens under the hood with regard to the pluggable-protocol framework. Normally, as a CORBA user, you would not be concerned with these details, since one of the great features of CORBA is that it hides all the implementation and mechanical details from you. As a pluggable-protocol developer, however, you need know what is happening inside the ORB.

There are three important parts within this discussion: the main program; the service-configurator file; and the command line arguments. The main program is from the Messenger example described in Chapter 3.

The service-configurator file for this discussion, svc.conf, contains the following line:

```
static Resource_Factory "-ORBProtocolFactory IIOP_Factory"
```

This line causes the ORB to use IIOP for communication between client and server.

The command line arguments for this example are:

```
-ORBListenEndpoint iiop://192.168.1.6:9999
```

The above option instructs the server-side code to set up the IIOP protocol to begin listening for requests at the given host IP address and port.

Here is the server’s main program:

```c
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main( int argc, char* argv[] )
{
    try {
```
// Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );

// Get reference to Root POA
CORBA::Object_var obj = orb->resolve_initial_references( "RootPOA" );
PortableServer::POA_var poa = PortableServer::POA::_narrow( obj.in() );

// Activate the POA via the POA Manager
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Create a servant
Messenger_i messenger_servant;

// Register the servant with the root POA, obtain its object
// reference, stringify and write the reference to a file.
PortableServer::ObjectId_var oid =
    poa->activate_object( &messenger_servant );
CORBA::Object_var messenger_obj = poa->id_to_reference( oid.in() );
CORBA::String_var str = orb->object_to_string( messenger_obj.in() );
std::ofstream iorFile( "Messenger.ior" );
iorFile << str.in() << std::endl;
iorFile.close();
std::cout << "IOR written to file Messenger.ior" << std::endl;

// Accept requests
orb->run();
orb->destroy();
}

catch( CORBA::Exception& ) {
    std::cerr << "Uncaught CORBA exception" << std::endl;
    return 1;
}

return 0;
}

We now consider certain lines of the above program to see how they relate to
the underlying pluggable-protocol framework.

Line 1 This is the standard call to CORBA::ORB_init() with the command line
arguments argc and argv passed as parameters. During this call, a number of
important actions occur with regard to the pluggable-protocol framework:

- During ORB initialization, the ORB processes the service-configuration file and instantiates the protocol-specific factory object, from which the
internal code is able to obtain other protocol-specific components that are used later for connection establishment.

In our example, we instruct the service configurator to instantiate and initialize a TAO_IIOP_Protocol_Factory object. This type of concrete factory class is one of the protocol-specific classes that each pluggable-protocol developer must define and implement. It follows the Abstract Factory pattern, as described in Design Patterns. It is from this concrete factory object that additional protocol-specific object instances are created.

In this case, the Abstract Factory is the TAO_Protocol_Factory base class as defined in Protocol_Factory.h in the $TAO_ROOT/tao directory. If the service-configurator file instructs the ORB initialization code to create additional protocol-specific factory objects, then these objects will be added to a collection of factory objects maintained by the ORB core. This collection plays the role of manager, with regard to the concrete factories, as described in the Manager pattern, found in Pattern Languages Of Program Design, Volume 3.

In later sections we discuss exactly how to implement the protocol-factory class so that it is compatible with the service configurator and other TAO internal classes.

- After creating the protocol factories in step 1, the ORB initialization code iterates through this collection of protocol factories and uses the concrete factories to create protocol-specific connector objects. It is the responsibility of these connector objects to handle the active connection establishment with any server-side code. After these objects are created, they are each stored in the TAO_Connector_Registry, which is later used by the ORB core to access these connector objects, and thereafter to actively establish connections with other applications. This scheme also follows the Manager pattern.

In our case the factory creates a TAO_IIOP_Connector object. We cover the details of the connector and other protocol-specific classes in greater detail later in this chapter.

**Line 2** Any server-side code requires that certain entities be created and initialized (i.e., the root POA). During this call to resolve_initial_references(), the server-side pluggable-protocol components are created, initialized, and set up to support server-side functionality.
In a manner similar to the handling of connectors, the server-side initialization code creates the acceptor registry, TAO_Acceptor_Registry, which is responsible for maintaining a set of protocol-specific acceptor objects. These acceptor objects are used by the ORB to passively accept connections from clients and to set up further message processing. As part of its initialization, the acceptor registry iterates through the set of protocol-specific factories created as a result of earlier ORB initialization.

The acceptor-registry code searches the list of factories to find a factory that has a prefix variable matching the one(s) specified on the command line argument that are part of the -ORBListenEndpoint option. In our example, the IIOP_Protocol_Factory code module initializes a prefix variable set to the string “iiop” (compared to the “iiop://” supplied in the listen endpoint option). Upon finding a factory with a matching prefix, the acceptor registry instructs the particular factory object to create a protocol-specific acceptor object and initialize it to listen at the specified listen endpoint. If more than one listen endpoint is specified on the command line, acceptors are created and initialized for each of these endpoints.

If no listen endpoint is specified on the command line, the acceptor-registry instructs each registered factory to open an acceptor using a default listen endpoint. Only acceptors designed to support default endpoints are opened this way.

The initialization of the acceptor object includes registering a service-handler object with the ORB’s core reactor. This service handler is called back upon the receipt of a client connection request. The dynamics of this activity are described fully by the Acceptor and Reactor patterns. As with the connector registry, upon successful creation and initialization of the protocol-specific acceptor object, the acceptor registry adds the newly-created acceptor to its collection as per the Manager pattern.

In our example, the acceptor-registry code uses the TAO_IIOP_Factory object to instantiate a TAO_IIOP_Acceptor.

**Line 3** Although object references are used by clients to invoke functions upon CORBA objects, object-reference creation is the responsibility of the server-side code, as described in *Advanced CORBA Programming with C++*. Object references for CORBA objects contain protocol-specific information. This information is encapsulated inside the protocol-specific *endpoint* and *profile* classes.
The endpoint class is an abstraction of the protocol-specific addressing scheme. In other words, an endpoint is a lightweight encapsulation of addressing information for a single acceptor endpoint, and it represents a single point of contact for the server. The endpoint is the smallest unit of addressing information necessary for a client to connect to a server.

The profile class is an abstraction of all the protocol-specific information that must be encoded in the IOR, which the client-side ORB can then decode in an effort to contact the servant. In other words, the profile class is an abstraction for representing object-location information. In the case of IIOP, the contents of the profile are based on the CORBA IOR definitions as outlined in the OMG GIOP and IIOP specifications. For all other protocols, the profile contains information specific to each protocol. A profile contains one or more endpoints (i.e., provides one or more ways to contact a server).

The POA that creates the object reference contains code that iterates through all concrete acceptors in the acceptor registry and tells each protocol-specific acceptor to create its profile information. After all acceptors have created profiles for each endpoint for each protocol, the POA has a collection of these profiles which it eventually encodes and places in the object reference.

In the case of IIOP, the classes involved with object reference creation are TAO_IIOP_Endpoint and TAO_IIOP_Profile. It is during the calls that create object references, such as id_to_reference(), _this(), and create_reference(), that these classes are instantiated and used to create the information needed by the object reference. This information is exported, then used by the client to contact CORBA servants via particular protocols.

This is the standard call to orb->run() that drives the ORB event loop. With regard to the pluggable-protocol framework, the function of orb->run() can be broken down into two parts: the code that is executed before any client requests arrive; and the code that is executed after a client connection request is made and the actual client request arrives at the server.

After the call to orb->run(), but before client requests arrive, the ORB waits to accept connection requests. The instantiated acceptors listen on endpoints, as dictated by the command line arguments. These acceptors have service handlers that are registered with the ORB reactor, so these service handlers are called back when connection requests come in, setting in motion the second part of the orb->run() call. Before client requests begin to arrive, no protocol-specific code is executed by the call to orb->run().
The next section outlines what happens after a client connection request arrives at the server side, then what happens after the actual client remote invocation arrives. All the participants of the pluggable-protocol framework collaborate with each other and with the ORB infrastructure to handle the client’s remote invocation. The following is an overview of the actions and collaborations that occur on the server side as the result of a client connection request and the subsequent client invocation on the server.

16.3.1.1 Handling the Connection Request

Prior to the arrival of a client connection request, a service handler is registered with the server-side reactor in the ORB core for the express purpose of being called back when a connection request arrives on a specific endpoint. Upon the arrival of a connection request from the client, this service handler is called back via its `handle_input()` function. This is the canonical behavior of the Acceptor pattern in conjunction with the Reactor pattern, described in *Pattern Oriented Software Architecture, Volume 2*. This `handle_input()` function is responsible for instantiating a protocol-specific connection handler and indirectly, a transport object that is thereafter responsible for handling the client request. Through the use of C++ templates, `handle_input()` is protocol independent, but capable of instantiating and invoking functions upon protocol-specific objects. In our example, `handle_input()` causes an `IIOP_Connection_Handler` and a `IIOP_Transport` object to be instantiated. These two objects work together to receive client requests, pass them up to the higher-level GIOP messaging layer (and eventually up to the servant), and send the reply to the client.

The main responsibilities of a transport object are to encapsulate a connection and to provide a transport-independent way to send and receive data. Since TAO is heavily based upon the reactor for most, if not all, of its I/O, the transport class is usually implemented with a helper connection handler that adapts the generic transport interface to a type that is compatible with the reactor framework. The transport object is the link from the pluggable-protocol layer to the higher-level pluggable messaging layer (GIOP handling code). While the transport provides a transport-independent way (i.e., an algorithm) to send and receive data, it uses the Template Method pattern, described in *Design Patterns*, to achieve protocol-specific functionality for certain parts of the algorithm by deferring some actions to protocol-specific derived classes. We cover this in more detail later in this chapter.
After instantiating the protocol-specific connection handler and transport, the `handle_input()` function invokes code that accepts the connection request into the newly-created protocol-specific connection handler. Ultimately, the acceptor (and the connector) are themselves factory objects responsible for creating and returning an I/O entity different from the one that was being used to establish the connection. This new I/O entity is used by the connection handler as the means to actually send and receive requests and replies. In other words, after the accept action occurs successfully, the connection handler has been configured with a stream-oriented handle (or protocol-specific entity) that is thereafter used to receive the client request and send the reply.

The `handle_input()` function activates the newly-created and initialized connection handler by calling the setup functions of the protocol-specific connection handler (i.e., `open()` ) and registering this new connection handler with the reactor, then returns control to the ORB event loop. The server-side ORB is now ready to receive remote client invocations on this connection.

### 16.3.1.2 Handling the Client Invocation

The arrival of a client request, after the connection establishment, results in the immediate callback to the protocol-specific connection-handler’s `handle_input()` function as per the Reactor pattern’s dynamics. The connection handler delegates the handling of the message to the transport object, which receives the data and passes the information to the GIOP messaging level of the ORB. The ORB then makes the upcall into the appropriate servant. The transport object collaborates with the messaging layer to get the reply from the servant and to get send it to the client in a protocol-specific manner. Additional requests over the same connection result in the reactor making a callback directly to the protocol-specific connection handler. This ends the cycle of receiving a request on the server side. We cover this scenario in more detail, showing code snippets of both ORB internal code and protocol-specific code, later in this chapter.

### 16.3.2 A Simple Example: Client Side

Now that we have covered the high-level collaborations on the server side, we present a similar description for the client side. The example we use is again the `Messenger` introduced in Chapter 3. The client-side main program from this example is shown below:

```c
#include "MessengerC.h"
```
#include <iostream>

int main (int argc, char* argv[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
        // Read and destringify the Messenger object's IOR.
        CORBA::Object_var obj = orb->string_to_object ("file://Messenger.ior");
        if (CORBA::is_nil(obj.in())) {
            std::cerr << "Could not get Messenger IOR." << std::endl;
            return 1;
        }
        // Narrow the IOR to a Messenger object reference.
        Messenger_var messenger = Messenger::_narrow (obj.in());
        if (CORBA::is_nil(messenger.in())) {
            std::cerr << "IOR was not a Messenger object reference." << std::endl;
            return 1;
        }
        CORBA::String_var message = CORBA::string_dup ("Hello!");
        messenger->send_message ("TAO User", "TAO Test", message.inout());
        // Print the Messenger's reply.
        std::cout << "Reply: " << msg.in() << std::endl;
        return 0;
    }
    catch (CORBA::Exception& ex) {
        std::cerr << "MessengerClient CORBA exception: " << ex << std::endl;
    }
    return 1;
}

We now consider particular lines of the above program to see how they relate to the underlying pluggable-protocol framework.

**Line 1** This is the identical call to CORBA::ORB_init() that is in the server-side code. The action of this function is the same as that on the server side. That is, the ORB initialization code creates protocol-specific factories as dictated by the contents of the service-configurator file, then uses these factory objects to create instances of protocol-specific connector objects that the ORB internals use to actively create connections with the server. As with the server-side
code, these connector objects are stored and managed in a connector-registry object.

**Line 2** This line brings the server-created object reference into the address space of the client. When this occurs (either through a call to the Naming Service or `string_to_object()`), a number of protocol-specific actions occur. First, the ORB connector registry uses the protocol-specific tag in the object reference information to get access to the protocol-specific connector. The registry then tells the connector to create a usable profile from the input IOR (which at this point has been put into an input CDR stream). The connector does this by instantiating a protocol-specific profile object, which decodes the input CDR stream.

Part of instantiating a profile object involves the instantiating of an endpoint object. The protocol-specific profile (`IIOP_Profile` in this example) decodes the CDR stream, then extracts the needed information and stores it in various member variables of the concrete profile object. The ORB internals use this decoded information to create a stub object through the stub factory. From this stub, the ORB code creates the necessary object reference that the client application code can use for invocations. After this whole process, the client application code can invoke operations on that object reference. Later in this chapter, we cover the details of the decoding that allows the client to use the object reference.

**Line 3** This is where the remote invocation is initiated. In the first protocol-related action that occurs, the connector registry is asked to make a connection with the remote server. The connector registry searches through its collection of protocol-specific connectors until it finds the one it needs (based on the tag value it has extracted from the endpoint information in the object reference), then delegates the active connection responsibilities to the protocol-specific connector component since it knows (or communicates with classes that know) how to make a connection using a particular protocol. If the connector does not already have an open connection to the server of interest (cached from an earlier invocation), it opens a new connection.

In the process of establishing a new connection with the server, the ORB code instantiates a protocol-specific connection handler and connects a new IO handle to it as described by the Connector pattern. This new connection handler is immediately activated (i.e., opened) and registered with the reactor. The stub and messaging code use the protocol-specific transport to send the request to the server. The transport object does this by delegating the send
operation to the connection-handler’s peer member. Once the send operation is complete, the messaging-layer code runs the reactor, ensuring that the reply may be received as a result of a the reactor callback to the protocol-specific connection handler that was registered with the reactor earlier.

Once the reply arrives from the server, the reactor code calls the `handle_input()` function of the protocol-specific connection handler (IIOP_Connection_Handler in this case). This connection handler delegates the responsibility of receiving the reply to the transport object (in this case an instance of TAO_IIOPTransport). As with the server side, the transport object specifies the steps and order in which the reply is received, but defers the protocol-specific steps to the overridden functions of the protocol-specific derived class. The reply data is then returned to the application through protocol-independent code.

**Line 4**
This line is a duplicate of line 3. The actions that are taken to carry out this invocation are the same as line 3, except that during the second invocation, the ORB checks to see if a connection exists to the server hosting the remote object and, if so, uses it instead of opening a new connection. Given that the overhead of holding a connection open is much less than the overhead of opening a new connection, the ORB holds open any connection as long as possible. However, it is possible that additional connections are created due to configuration options set in the client strategy factory (see Chapter 22). For instance, if the multiplexing strategy is set to `exclusive`, and a connection is currently busy, then a new connection is made for the new request.

This completes the overview of both the client- and server-side protocol-related actions. The rest of this chapter provides more detail about how to define and implement your protocol-specific classes to provide seamless integration with the dynamics and context of the TAO pluggable-protocol framework.

### 16.4 Details of the Pluggable-Protocol Framework

There are a small number of classes that you need to define and implement to enable your protocol to work with the TAO pluggable-protocol framework. This framework allows you to add new protocols without changing TAO itself by providing an interface between TAO and your custom protocol. As long as your protocol classes conform to expectations and patterns of the framework,
the task of producing a TAO pluggable protocol can be as simple as following a recipe. Using existing protocols as models for your protocol can make your job much easier.

The TAO pluggable-protocol framework provides abstract base classes from which your protocol-specific classes must derive. Table 16-1 shows the framework components that must be implemented for each protocol, the base classes that define each interface, and the header files in which the base classes are defined.

### Table 16-1 Pluggable-Protocol Classes and Files

<table>
<thead>
<tr>
<th>Component</th>
<th>Base Class</th>
<th>Base-Class Header File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptor</td>
<td>TAO_Acceptor</td>
<td>$TAO_ROOT/tao/Transport_Acceptor.h</td>
</tr>
<tr>
<td>Connector</td>
<td>TAO_Connector</td>
<td>$TAO_ROOT/tao/Transport_Connector.h</td>
</tr>
<tr>
<td>Connection Handler</td>
<td>TAO_Connection_Handler</td>
<td>$TAO_ROOT/tao/Connection_Handler.h</td>
</tr>
<tr>
<td>Endpoint</td>
<td>TAO_Endpoint</td>
<td>$TAO_ROOT/tao/Endpoint.h</td>
</tr>
<tr>
<td>Profile</td>
<td>TAO_Profile</td>
<td>$TAO_ROOT/tao/Profile.h</td>
</tr>
<tr>
<td>Protocol Factory</td>
<td>TAO_Protocol_Factory</td>
<td>$TAO_ROOT/tao/Protocol_Factory.h</td>
</tr>
<tr>
<td>Transport</td>
<td>TAO_Transport</td>
<td>$TAO_ROOT/tao/Transport.h</td>
</tr>
</tbody>
</table>

The source code for the derived types is typically divided into several files, which are usually named for the protocol and component expressed within. For instance, IIOP-related classes are in files such as IIOP_Acceptor.h and IIOP_Connector.h, located in the $TAO_ROOT/tao directory. Other protocol implementations, such as UIOP and SHMIOP, are located in the $TAO_ROOT/tao/Strategies directory. SSLIOP implementations are located in the $TAO_ROOT/orbsvcs/orbsvcs/SSLIOP directory.

The rest of this section is devoted to describing the interfaces that must be implemented by any pluggable protocol. As is typical with any complex programming exercise, simply fulfilling the interface is not sufficient for providing a robust implementation. The details of some of the work are dependent on the particulars of the protocol. To help you understand the nature of such details, we present samples of the TAO-supplied pluggable protocols for illustration. Refer to the directories listed above for the full source code for these existing protocols.
Developing Pluggable Protocols

There are seven fundamental classes included in a TAO pluggable protocol. These classes are described in general below and in detail in the following sections:

- **The Acceptor** provides the passive endpoint in servers that opens new connections as requested by clients. Within the context of the pluggable-protocol framework, there are several distinct acceptor classes:
  - The abstract base-class `TAO_Acceptor` serves as the interface supplied by the framework. In the following discussion it is referred to as the “the TAO acceptor.”
  - A protocol-specific acceptor class that you provide to implement this interface. It is referred to as the “pluggable acceptor,” or simply, “your acceptor.” The name “TAO_*_Acceptor” is used to refer to any protocol-specific acceptor type, of which there are several.
  - The class that is actually responsible for accepting new connections. This class is commonly the `ACE_Strategy_Acceptor` template class. It is referred to as the “base acceptor.”
  - In many protocols, there is another acceptor class, which is one of the template parameters to the `ACE_Strategy_Acceptor` class, and is referred to as the “peer acceptor.” This acceptor provides an extra level of indirection that leverages the existing ACE framework classes that assist in implementing the protocol-specific behavior. Examples of these classes include `ACE_SOCK_Acceptor` (for IIOP) and `ACE_MEM_Acceptor` (for SHMIOP). Later sections show how to leverage the existing ACE framework classes to support your protocol.

- **The Connector** provides an active endpoint for creating connections. Within the context of the pluggable-protocol framework, there are several distinct connector classes:
  - The abstract base-class `TAO_Connector`, which serves as the interface supplied by the framework. In the following discussion it is referred to as “the TAO connector.”
  - A protocol-specific connector class that you provide to implement this interface as dictated by the TAO-connector base class. It is referred to either as the “pluggable connector” or “your connector.”
The name “TAO_*_Connector” is used to refer to any protocol-specific connector type, of which there are several.

- The class that is responsible for establishing new connections. This class is commonly the ACE_Strategy_Connector template class, and is referred to as “the base connector.”

- In many protocols, there is another acceptor class, which is one of the template parameters to the ACE_Strategy_Connector class, and is referred to as the “peer connector.” This connector provides an extra level of indirection that leverages the existing ACE framework classes that assist in implementing the protocol-specific behavior. Examples of these classes include ACE_SOCK_Connector (for IIOP) and ACE_MEM_Connector (for SHMIOP). Later sections show how to leverage the existing ACE framework classes to support your protocol.

- **The Connection Handler** manages the interface between TAO and the actual communication protocol as managed by the transport abstraction. Your protocol implementation must include a subclass of the base-class TAO_Connection_Handler. There is only one connection-handler class that is used by both clients and servers.

- **The Endpoint** represents a single point of contact for the server, and is the smallest unit of addressing information necessary for a client to connect to a server. Your protocol implementation must include a subclass of the TAO_Endpoint base class and must contain the addressing information specific to your custom protocol.

- **The Profile** is used to communicate protocol information between the various objects within an application. This information is encoded within an IOR that is used to communicate endpoint information from servers to clients. Your protocol implementation must include a subclass of the TAO_Profile base class. It must also contain the necessary protocol-specific information to be encoded into a IOR, exported to clients, decoded by clients, and used to contact the server via the custom protocol.

- **The Protocol Factory** is responsible for constructing acceptors and connectors. Your protocol implementation must include a subclass of the TAO_Protocol_Factory base class.
The Transport works in conjunction with the connection handler to provide an interface to the higher-level messaging layer of TAO, and a level of indirection down to the send and receive mechanics of the concrete protocol.

The following sections show the details of each of these interfaces, and give examples and explanations of concrete protocol implementations of them.

16.4.1 The Acceptor

The acceptor is responsible for opening and managing the passive endpoint of a connection. The pluggable-protocol framework supplies the TAO_Acceptor abstract base class, which defines the interface through which the ORB accesses the protocol-specific acceptor.

TAO constructs an acceptor object for each endpoint provided to the ORB at initialization. If no endpoints are explicitly provided to the ORB, an acceptor is constructed for a listed protocol, as long as the protocol factory has been instantiated and this protocol factory allows for implicit activation of the protocol.

Once the acceptor is activated, it is able to accept new connections, and provide the required transport and connection-handler objects to the ORB that enable communication with the new client.

In addition to initializing endpoints and accepting new connections, the acceptor is also responsible for certain ORB utility functions, such as determining if a desired connection is collocated, adding endpoints to existing profiles, and generating the profiles supplied in object references.

Acceptors are owned by the acceptor registry, which is created by the TAO_Default_Resource_Factory. It is the responsibility of the pluggable acceptor to create a passive endpoint for a provided (or default) address, and to supply a profile representing that endpoint as needed.

Here is the definition for the abstract base class for pluggable acceptors:

```cpp
class TAO_Export TAO_Acceptor
{
public:
    TAO_Acceptor (CORBA::ULong tag);
    virtual ~TAO_Acceptor (void);

    CORBA::ULong tag (void) const;
    virtual int open (TAO_ORB_Core* orb_core,
```
ACE_Reactor* reactor,
int version_major,
int version_minor,
const char* address,
const char* options = 0) = 0;
virtual int open_default (TAO_ORB_Core*,
    ACE_Reactor* reactor,
    int version_major,
    int version_minor,
    const char* options = 0) = 0;
virtual int close (void) = 0;
virtual int create_profile (const TAO_ObjectKey &object_key,
    TAO_MProfile &mprofile,
    CORBA::Short priority) = 0;
virtual int is_collocated (const TAO_Endpoint* endpoint) = 0;
virtual CORBA::ULong endpoint_count (void) = 0;
virtual int object_key (IOP::TaggedProfile &profile,
    TAO_ObjectKey& key) = 0;
private:
    CORBA::ULong tag_;}

As you can see from the class definition, the base class only provides implementations for the constructor, destructor, and the tag accessor function. When implementing the protocol-specific concrete acceptor, you are responsible for providing the concrete overrides for the pure virtual functions specified in the base class. These functions are used by the framework to open and close accepting connections, and for composing profiles supplied to clients as part of interoperable object references.

For completeness, and as an example, we include the class definition for the TAO_UIOP_Acceptor:

class TAO_Strategies_Export TAO_UIOP_Acceptor : public TAO_Acceptor
{
public:
    TAO_UIOP_Acceptor (CORBA::Boolean flag = 0);
    virtual ~TAO_UIOP_Acceptor (void);

    typedef ACE_Strategy_Acceptor<TAO_UIOP_Connection_Handler, ACE_LSOCK_ACCEPTOR>
    TAO_UIOP_BASE_ACCEPTOR;
    typedef TAO_Creation_Strategy<TAO_UIOP_Connection_Handler>
    TAO_UIOP_CREATION_STRATEGY;
    typedef TAO_Concurrency_Strategy<TAO_UIOP_Connection_Handler>
    TAO_UIOP_CONCURRENCY_STRATEGY;
    typedef TAO_Accept_Strategy<TAO_UIOP_Connection_Handler, ACE_LSOCK_ACCEPTOR>
    TAO_UIOP_ACCEPT_STRATEGY;
Developing Pluggable Protocols

virtual int open (TAO_ORB_Core* orb_core,
                ACE_Reactor* reactor,
                int version_major,
                int version_minor,
                const char* address,
                const char* options = 0);
virtual int open_default (TAO_ORB_Core* orb_core,
                          ACE_Reactor* reactor,
                          int version_major,
                          int version_minor,
                          const char* options = 0);
virtual int close (void);
virtual int create_profile (const TAO_ObjectKey& object_key,
                            TAO_MProfile &mprofile,
                            CORBA::Short priority);
virtual int is_collocated (const TAO_Endpoint* endpoint);
virtual CORBA::ULong endpoint_count (void);
virtual int object_key (IOP::TaggedProfile& profile,
                        TAO_ObjectKey& key);

private:
    int open_i (const char* rendezvous,
                ACE_Reactor* reactor);
    void rendezvous_point (ACE_UNIX_Addr&, const char* rendezvous);
    int parse_options (const char* options);
    int init_uiop_properties (void);
    int create_new_profile (const TAO_ObjectKey& object_key,
                            TAO_MProfile& mprofile,
                            CORBA::Short priority);
    int create_shared_profile (const TAO_ObjectKey& object_key,
                               TAO_MProfile& mprofile,
                               CORBA::Short priority);

private:
    TAO_UIOP_BASE_ACCEPTOR base_acceptor_;
    TAO_UIOP_CREATION_STRATEGY* creation_strategy_;
    TAO_UIOP_CONCURRENCY_STRATEGY *concurrency_strategy_;
    TAO_UIOP_ACCEPT_STRATEGY* accept_strategy_;
    TAO_GIOP_Message_Version version_;
    TAO_ORB_Core* orb_core_;
    bool unlink_on_close_;
    const bool lite_flag_;
    TAO_UIOP_Properties uiop_properties_;}

16.4.1.1 The Acceptor Typedefs and Strategies

The following typedefs are in the concrete acceptor class definition:
typedef ACE_Strategy_Acceptor<TAO_UIOP_Connection_Handler, ACE_LSOCK_ACCEPTOR> TAO_UIOP_BASE_ACCEPTOR;

typedef TAO_Creation_Strategy<TAO_UIOP_Connection_Handler> TAO_UIOP_CREATION_STRATEGY;

typedef TAO_Concurrency_Strategy<TAO_UIOP_Connection_Handler> TAO_UIOP_CONCURRENCY_STRATEGY;

typedef TAO_Accept_Strategy<TAO_UIOP_Connection_Handler, ACE_LSOCK_ACCEPTOR> TAO_UIOP_ACCEPT_STRATEGY;

All protocols currently provided with TAO leverage existing generic code to carry out their responsibilities. As you can see from the class definition, these typedefs are used to define the types of strategy member variables as in:

TAO_UIOP_BASE_ACCEPTOR base_acceptor_;  
TAO_UIOP_CREATION_STRATEGY* creation_strategy_;  
TAO_UIOP_CONCURRENCY_STRATEGY* concurrency_strategy_;  
TAO_UIOP_ACCEPT_STRATEGY* accept_strategy_;
Figure 16-1 is a UML diagram of the acceptor's relationship with its strategies.

Figure 16-1 Acceptor relationships

We now examine each of these typedefs in detail.

Strategy-Acceptor Template

Synopsis

```
template <class SVC_HANDLER, ACE_PEER_ACCEPTOR_1>
class ACE_Strategy_Acceptor
  : public ACE_Acceptor <SVC_HANDLER, ACE_PEER_ACCEPTOR_2>
```

This class is a generic and strategized version of the Acceptor pattern. It is a factory for creating a service handler, accepting connections from peers, and activating that service handler. Since this is a template, the particular service handler is bound at compile time. It performs these actions on the connection handler through the use of various strategy classes which are configured at run time. The two template parameters that are used to define this class are:

1. the type of service handler to be instantiated, accepted into, then activated.
2. the stream abstraction for the low-level I/O mechanism for use with the protocol.

The ACE_Strategy_Acceptor uses a combination of the Template Method pattern and the Strategy pattern to fulfill its role in the Acceptor pattern. Its base class has a handle_input() function which gets called back by the reactor framework when a connection request arrives from a client. The base class establishes the steps needed to comply with this connection request. As the Template Method pattern describes, the order of these steps is mandated by the base-class function, but their implementation is deferred to the derived class ACE_Strategy_Acceptor, which achieves its functionality by delegating its actions to specific strategy classes. The steps prescribed by the function handle_input() are:

1. make_svc_handler()
2. accept_svc_handler()
3. activate_svc_handler()

Below is the key template function with the key steps highlighted that put the protocol-specific connection handler into play:

```cpp
template <class SVC_HANDLER, ACE_PEER_ACCEPTOR_1> int
ACE_Acceptor<SVC_HANDLER, ACE_PEER_ACCEPTOR_2>::handle_input
(ACE_HANDLE listener)
{
    ACE_TRACE ("ACE_Acceptor<SVC_HANDLER, ACE_PEER_ACCEPTOR_2>::handle_input");
    ACE_Handle_Set conn_handle;

    // Default is "timeout (0, 0)," which means "poll."
    ACE_Time_Value timeout;

    do {
        // Create a service handler, using the appropriate creation
        // strategy.
        SVC_HANDLER *svc_handler = 0;

        if (this->make_svc_handler (svc_handler) == -1) {
            if (ACE::debug () > 0)
                ACE_DEBUG ((LM_DEBUG,
                    ACE_LIB_TEXT ("%p\n"),
                    ACE_LIB_TEXT ("make_svc_handler")));
            return 0;
        }

        // Accept connection into the Svc_Handler.
```
else if (this->accept_svc_handler (svc_handler) == -1) {
    return 0;
}

else if (this->activate_svc_handler (svc_handler) == -1) {
    if (ACE::debug () > 0)
        ACE_DEBUG ((LM_DEBUG,
            ACE_LIB_TEXT ("%p\n"),
            ACE_LIB_TEXT ("activate_svc_handler"));
    return 0;
}

    conn_handle.set_bit (listener);
}

while (this->use_select_ && ACE_OS::select (int (listener) + 1,
    conn_handle, 0, 0, &timeout) == 1);
return 0;

The `handle_input()` function execution begins in the `ACE_Acceptor` base class by invoking the above three functions on itself, but their execution is deferred while the derived `ACE_Strategy_Acceptor` forwards them onto the various strategy objects for actual execution.

These concrete strategy classes are determined by the other typedefs and are discussed below.

**Creation-Strategy Template**

**Synopsis**

template <class SVC_HANDLER>

class TAO_Creation_Strategy : public ACE_Creation_Strategy<SVC_HANDLER>

The creation-strategy class is instantiated with the concrete protocol-specific connection handler as a template parameter. The `TAO_Creation_Strategy` subclass overrides the `make_svc_handler()` function and adds cache-purging functionality to the default behavior, as shown in the following code:

```cpp
template <class SVC_HANDLER> int
    TAO_Creation_Strategy<SVC_HANDLER>::make_svc_handler (SVC_HANDLER* &-sh)
    {
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
if (sh == 0) {
    // Purge connections (if necessary)
    this->orb_core_->lane_resources ()->transport_cache ()->purge ();

    ACE_NEW_RETURN (sh, 
        SVC_HANDLER (this->orb_core_, 
            this->lite_flag_, 
            this->arg_), 
        -1);
}

return 0;
}

Accept-Strategy Template

Synopsis

template <class SVC_HANDLER, ACE_PEER_ACCEPTOR_1> 
class TAO_Accept_Strategy :
    public ACE_Accept_Strategy<SVC_HANDLER,ACE_PEER_ACCEPTOR_2>

The TAO_Accept_Strategy class is instantiated with the concrete protocol-specific connection handler and the concrete protocol-specific stream acceptor as template parameters. The accept_svc_handler() function of this class simply calls the base-class functionality, which accepts the new connection into the protocol-specific connection handler as shown below:

template <class SVC_HANDLER, ACE_PEER_ACCEPTOR_1> int 
TAO_Accept_Strategy<SVC_HANDLER, ACE_PEER_ACCEPTOR_2>::accept_svc_handler 
   (SVC_HANDLER* svc_handler) 
   { 
       return ACCEPT_STRATEGY_BASE::accept_svc_handler (svc_handler);
   }

The base-class functionality is as follows:

template <class SVC_HANDLER, ACE_PEER_ACCEPTOR_1> int 
ACE_Accept_Strategy<SVC_HANDLER, ACE_PEER_ACCEPTOR_2>::accept_svc_handler 
   (SVC_HANDLER* svc_handler) 
   {
       int reset_new_handle = this->reactor_->uses_event_associations ();

       if (this->peer_acceptor_.accept (svc_handler->peer (), // stream 
                                       0, // remote address 
                                       0, // timeout 
                                       1, // restart 
                                       reset_new_handle) // reset new handler
           return 0;
   }
Developing Pluggable Protocols

Concurrency-Strategy Template

Synopsis
template <class SVC_HANDLER>
class TAO_Concurrency_Strategy :
   public ACE_Concurrency_Strategy<SVC_HANDLER>

The concurrency-strategy class is instantiated with the concrete protocol-specific connection handler as a template parameter. It overrides the activate_svc_handler() function, adds caching functionality, and turns the connection handler into an active object as directed by the service configuration option –ORBWaitStrategy. Finally, the activate_svc_handler() registers the connection handler with the reactor:

template <class SVC_HANDLER> int
TAO_Concurrency_Strategy<SVC_HANDLER>::activate_svc_handler (SVC_HANDLER* sh, void* arg)
{
   if (this->ACE_Concurrency_Strategy<SVC_HANDLER>::activate_svc_handler (sh, arg) == -1)
      return -1;

   // The service handler has been activated. Now cache the handler.
   if (sh->add_transport_to_cache () == -1) {
      ACE_ERROR ((LM_ERROR,
         ACE_TEXT ("(%P|%t) Could not add the handler to Cache \n")));
   }

   TAO_Server_Strategy_Factory* f =
      this->orb_core_->server_factory ();

   // thread-per-connection concurrency model
   if (f->activate_server_connections ()
      return sh->activate (f->server_connection_thread_flags (),
         f->server_connection_thread_count ());
To either add or override functionality as a custom protocol developer, you may need to use different strategies and/or write your own. When doing so, be sure to conform to the conventions followed by the existing protocols and classes.

### 16.4.1.2 Member Functions in Detail

Each of the functions defined by the acceptor interface is described below.

**Pluggable-Acceptor Constructor**

The pluggable acceptor is constructed by a protocol-specific factory. Since the protocol factory is under your control, the signature of your concrete acceptor is entirely up to you. Your concrete acceptor constructor must, however, call the `TAO_Acceptor` constructor explicitly in the initialization list, and supply it with the tag value reserved for your protocol. The concrete constructor should thereafter only initialize state. Any initialization related to connection establishment, such as address assignment, should be deferred until the `open()` function is called. An example of the `TAO_UIOP_Acceptor` constructor (the acceptor for the Unix Domain Sockets pluggable protocol) is shown below:

```cpp
TAO_UIOP_Acceptor::TAO_UIOP_Acceptor (CORBA::Boolean flag)
  : TAO_Acceptor (TAO_TAG_UIOP_PROFILE),
    base_acceptor_ (),
    creation_strategy_ (0),
    concurrency_strategy_ (0),
    accept_strategy_ (0),
    version_ (TAO_DEF_GIOP_MAJOR, TAO_DEF_GIOP_MINOR),
    orb_core_ (0),
    unlink_on_close_ (true),
    lite_flag_ (flag)
{
}
```

We initialize the base constructor with `TAO_TAG_UIOP_PROFILE`, the unique identifier for UIOP profiles. The `base_acceptor_` member of the derived class is actually a protocol-specific class (in particular, a template generic
class parameterized with a protocol-specific template parameter—see the typedef in the class definition for TAO_UIOP_Acceptor). The initialization of the base_acceptor is discussed further in the description of the open() function. The strategy objects in the constructor’s initialization list relate to the base_acceptor. It is very likely that your protocol’s acceptor will use these same strategy objects. Other common attributes are the version and the reference to the ORB core. The last attributes shown above are specific to UIOP.

The acceptor passes the flag argument later to strategies that it instantiates. In the case of UIOP and IIOP, the acceptor and connector are used in two forms: one that supports the standard CORBA messaging protocol GIOP; and the other that supports a TAO-specific optimization of GIOP, known as GIOP “lite.” There are different protocol factories used to differentiate these two messaging protocols, but they use the same acceptor, connector, etc.

**Pluggable-Acceptor Destructor**

The base-class destructor serves no purpose other than to define the virtual function, so that proper clean up occurs when deletion of a derived object occurs through the use of a base-class pointer. In most cases, your protocol-specific acceptor will own heap-allocated objects, such as the various strategy objects used to control the base_acceptor.

In this example, the close() function is also called by the destructor to ensure that the strategy objects are not in use when they are deleted:

```
TAO_UIOP_Acceptor::~TAO_UIOP_Acceptor (void)
{
    // Make sure we are closed before we start destroying the
    // strategies.
    this->close ();

    delete this->creation_strategy_;
    delete this->concurrency_strategy_;
    delete this->accept_strategy_;
}
```

**Pluggable-Acceptor open() Function**

**Synopsis**

```
int open (TAO_ORB_Core* orb_core,
          ACE_Reactor* reactor,
          int version_major,
          int version_minor,
```
16.4 Details of the Pluggable-Protocol Framework

This function is responsible for initializing the acceptor, parsing any options, and opening the endpoint. The values supplied as arguments to `open()` come from the `-ORBListenEndpoints` option, described in 19.6. It is the responsibility of `open()` to validate the provided address, version number, and options, then begin listening for connections on that endpoint. For example:

```
-ORBListenEndpoints iio://1.2@www.theaceorb.com:12345/opt=foo
```

results in the following call to `open()`:

```
open(orb_core,1,2,"www.theaceorb.com:12345","opt=foo");
```

When multiple endpoints are specified, TAO constructs and opens a new acceptor for each one.

We now examine the `TAO_UIOP_Acceptor::open()` function behavior:

```c
int
TAO_UIOP_Acceptor::open (TAO_ORB_Core* orb_core,
                           ACE_Reactor* reactor,
                           int major,
                           int minor,
                           const char* address,
                           const char* options)
{
    this->orb_core_ = orb_core;

    if (address == 0)
        return -1;

    if (major >= 0 && minor >= 0)
        this->version_.set_version (ACE_static_cast (CORBA::Octet, major),
                                    ACE_static_cast (CORBA::Octet, minor));

    // Parse options
    if (this->parse_options (options) == -1)
        return -1;
    else
        return this->open_i (address, reactor);
}
```
There are three operations performed by `open()` for this protocol:

1. The version number is initialized with the provided major and minor numbers.
2. The option string is parsed. The parsing is handled by an internal function that is not specified as part of the TAO_Acceptor interface, since options are unique to a particular protocol.
3. The internal `open_i()` function is called. This function is defined for TAO_UIOP_Acceptor, but is not part of the acceptor interface.

Parsing the options and performing the internal `open_i()` are operations that are common to both `open()` and `open_default()`.

The `open_i()` function of the TAO_UIOP_Acceptor does more than simply open the acceptor socket. As with all pluggable acceptors, this function is responsible for completing the initialization of the protocol-specific base acceptor, creating the associated strategy objects, and making sure no errors occur. As mentioned earlier in this chapter, the TAO acceptor delegates the connection-establishment responsibility to a base acceptor (in this case within the ACE code). In opening the base acceptor, the TAO acceptor provides it with the endpoint address and strategy objects.

The `open_i()` implementation is shown below:

```cpp
TAO_UIOP_Acceptor::open_i (const char* rendezvous,
                          ACE_Reactor* reactor)
{
  ACE_NEW_RETURN (this->creation_strategy_,
                  TAO_UIOP_CREATION_STRATEGY (this->orb_core_,
                                             &(this->uiop_properties_),
                                             this->lite_flag_),
                                             -1);

  ACE_NEW_RETURN (this->concurrency_strategy_,
                  TAO_UIOP_CONCURRENCY_STRATEGY (this->orb_core_),
                                             -1);

  ACE_NEW_RETURN (this->accept_strategy_,
                  TAO_UIOP_ACCEPT_STRATEGY (this->orb_core_),
                                             -1);

  ACE_UNIX_Addr addr;

  this->rendezvous_point (addr, rendezvous);

  if (this->base_acceptor_.open (addr,
                                 reactor,
                                 ...));
}
```
16.4 Details of the Pluggable-Protocol Framework

In the above code, you can first see the instantiation of the various strategy objects as dictated by the typedefs in the header files, then the open() call on the base ACE_Strategy_Acceptor. The strategy acceptor is portable across all protocols, so any pluggable acceptor uses the same form of open() on the base acceptor. In the above example, the code surrounding open() is specific to UNIX Domain Sockets.

Pluggable-Acceptor open_default() Function

Synopsis

```c
int open_default (TAO_ORB_Core*,
                 ACE_Reactor* reactor,
                 int major,
                 int minor,
                 const char* options = 0)
```

There are many times when a specific address is not supplied, yet an endpoint for a protocol is still desired. To meet this requirement, TAO_Acceptor provides the open_default() function. It is the responsibility of the
pluggable acceptor to generate an appropriate address to be used as an endpoint.

The open_default() function is invoked whenever the -ORBListenEndpoints option is passed a value that specifies a protocol but supplies no address. For example:

```
-ORBListenEndpoints iiop://opt1=foo;uiop://;shmiop://
```

This endpoint argument instructs TAO to create an acceptor for IIOP, UIOP, and SHMIOP. For each of these protocols, a specific endpoint is not supplied. As a result, the acceptor creates a default endpoint. The IIOP endpoint has an additional option, demonstrating how to supply options for the argument to open_default().

The following code shows the open_default() function from the UIOP acceptor. Just as open() performs a few initialization operations before calling the internal open_i() function, open_default() must set the major and minor version numbers, create the address, and parse the option string before calling open_i():

```c
int
TAO_UIOP_Acceptor::open_default (TAO_ORB_Core* orb_core,
      ACE_Reactor* reactor,
      int major,
      int minor,
      const char* options)
{
  this->orb_core_ = orb_core;

  if (major >=0 && minor >= 0)
    this->version_.set_version (ACE_static_cast (CORBA::Octet,
                   major),
                   ACE_static_cast (CORBA::Octet,
                   minor));

  // Parse options
  if (this->parse_options (options) == -1)
    return -1;

  ACE_Auto_String_Free tempname (ACE_OS::tempnam (0,
                                     "TAO");

  if (tempname.get () == 0)
    return -1;
```
return this->open_i (tempname.get (),
    reactor);
}

In this example of open_default(), the UIOP-specific properties, and the major and minor version are set. The options string is parsed and validated. Then an arbitrary rendezvous point is created using ACE_OS::tempnam(), since UNIX domain sockets use file-system entries as endpoints. Finally, the generated name is supplied to open_i() to complete the initialization of the acceptor. As this example shows, the concrete acceptor’s implementation is under your control, and is the place where you can add protocol-specific initialization and behavior.

**Pluggable-Acceptor close() Function**

**Synopsis**

```c
int close(void)
```

The close() function is called when the acceptor is to be shut down. At a minimum, you should call close() on your underlying base acceptor, but additional protocol-specific steps may be necessary to clean up persistent resources as shown below with UIOP:

```c
int
TAO_UIOP_Acceptor::close (void)
{
    if (this->unlink_on_close_)
    {
        ACE_UNIX_Addr addr;

        if (this->base_acceptor_.acceptor ().get_local_addr (addr) == 0)
            (void) ACE_OS::unlink (addr.get_path_name ());

        this->unlink_on_close_ = 0;
    }

    return this->base_acceptor_.close ();
}
```

**Pluggable-Acceptor create_profile() Function**

**Synopsis**

```c
int create_profile (const TAO_ObjectKey& object_key,
    TAO_MProfile& mprofile,
    CORBA::Short priority)
```
When TAO supplies an IOR for an object, it must include profiles in that IOR for all of the server’s endpoints. This is achieved by requesting from the acceptor registry a TAO_MProfile object for a given object key. This object is a container of multiple profile objects with one or more endpoints per profile. Because the type of the profile is unique to the protocol it represents, it is the responsibility of the pluggable acceptor to generate the appropriate profile object. When TAO is built with the RT-CORBA features enabled (the default), only one profile is created for each protocol. Non-RT-CORBA ORBs will try to create shared profiles when “-ORBUseSharedProfiles 1” is passed to ORB_init(). Subsequent endpoints with the same protocol simply add their endpoints to the profile. The acceptor uses the priority value to determine whether endpoints of the same protocol should share the same profile or exist in a new profile. The UIOP example is shown below:

```cpp
int
TAO_UIOP_Acceptor::create_profile (const TAO_ObjectKey& object_key,
    TAO_MProfile& mprofile,
    CORBA::Short priority)
{
    // Check if multiple endpoints should be put in one profile or
    // if they should be spread across multiple profiles.
    if (priority == TAO_INVALID_PRIORITY)
        return this->create_new_profile (object_key,
            mprofile,
            priority);
    else
        return this->create_shared_profile (object_key,
            mprofile,
            priority);
}
```

Not all existing protocol-specific acceptors honor the ORB’s shared profile flag. The UIOP acceptor shown above does not. The IIOP acceptor does. In that class, the create_profile() method extends the test for shared profiles as shown here:

```cpp
if (priority == TAO_INVALID_PRIORITY &&
    this->orb_core_->orb_params()->shared_profile() == 0)
```

The UIOP acceptor’s create_shared_profile() implementation begins by looking through the existing profiles for one of the correct type:
16.4 Details of the Pluggable-Protocol Framework

```cpp
tao::UIOP_Acceptor::create_shared_profile (const TAO_ObjectKey& object_key,
                                           TAO_MProfile& mprofile,
                                           CORBA::Short priority)
{
  TAO_Profile* pfile = 0;
  TAO_UIOP_Profile* uiop_profile = 0;
  // First see if <mprofile> already contains a UIOP profile.
  for (TAO_PHandle i = 0; i != mprofile.profile_count (); ++i) {
    pfile = mprofile.get_profile (i);
    if (pfile->tag () == TAO_TAG_UIOP_PROFILE) {
      uiop_profile = ACE_dynamic_cast (TAO_UIOP_Profile*,
                                       pfile);
      break;
    }
  }
  if (uiop_profile == 0) {
    // If <mprofile> doesn't contain UIOP_Profile, we need to create
    // one.
    return create_new_profile (object_key,
                               mprofile,
                               priority);
  }
  If an existing profile of the correct type is found, an endpoint object of the
  appropriate type is created and added to the profile.
  else {
    // A UIOP_Profile already exists - just add our endpoint to it.
    ACE_UNIX_Addr addr;
    if (this->base_acceptor_.acceptor ().get_local_addr (addr) == -1)
      return 0;
    TAO_UIOP_Endpoint* endpoint = 0;
    ACE_NEW_RETURN (endpoint,
                    TAO_UIOP_Endpoint (addr),
                    -1);
    endpoint->priority (priority);
    uiop_profile->add_endpoint (endpoint);
  }
```

If an existing profile of the correct type is not found, one of the correct type is
created and initialized with the acceptor’s endpoint information by using the
TAO_UIOP_Acceptor::create_new_profile() function. This function
is discussed below.
return 0;
}

The UIOP acceptor’s create_new_profile() function demonstrates how to create and initialize a profile. First, initial error conditions are tested, making sure that this acceptor has a valid address and that mprofile has space for this acceptor’s profile:

```c
int TAO_UIOP_Acceptor::create_new_profile (const TAO_ObjectKey& object_key,
                                         TAO_MProfile& mprofile,
                                         CORBA::Short priority)
{
    ACE_UNIX_Addr addr;

    if (this->base_acceptor_.acceptor ().get_local_addr (addr) == -1)
        return 0;

    int count = mprofile.profile_count ();
    if ((mprofile.size () - count) < 1
        && mprofile.grow (count + 1) == -1)
        return -1;

    Next, a profile object is generated, initialized with the priority and handed to mprofile.

    TAO_UIOP_Profile* pfile = 0;
    ACE_NEW_RETURN (pfile,
                    TAO_UIOP_Profile (addr,
                                     object_key,
                                     this->version_,
                                     this->orb_core_),
                    -1);
    pfile->endpoint ()->priority (priority);

    if (mprofile.give_profile (pfile) == -1) {
        pfile->_decr_refcnt ();
        pfile = 0;
        return -1;
    }
```

Finally, if the server is configured to generate standard profile components, these must be added to the profile.
16.4 Details of the Pluggable-Protocol Framework

// Do not add any tagged components to the profile if configured
// by the user not to do so, or if an UIOP 1.0 endpoint is being
// created (IIOP 1.0 did not support tagged components, so we follow
// the same convention for UIOP).
if (this->orb_core_->orb_params ()->std_profile_components () == 0
    || (this->version_.major == 1 && this->version_.minor == 0))
    return 0;

    pfile->tagged_components ().set_orb_type (TAO_ORB_TYPE);

    TAO_Codeset_Manager *csm = this->orb_core_->codeset_manager();
    if (csm)
        csm->set_codeset (pfile->tagged_components());

    return 0;
}

The codeset manager, if present, will supply an appropriate codeset
component only if standard profile components are supported. The codeset
manager must be tested for nil because its existence is optional. However
TAO may be compiled to not perform codeset negotiation, or this behavior
may be turned off via the ORB_init() option “-ORBNegotiateCodesets
0.”

Pluggable-Acceptor is_collocated() Function

Synopsis
int is_collocated (const TAO_Endpoint* endpoint)

Collocation provides an opportunity for an immense improvement in
performance, because invocations may not have to be transmitted through a
transport layer between the client and the server.

The determination of collocated status is made when the client is converting
an IOR to an object. During this process, the multi-profile object is extracted
from the IOR and passed to the acceptor registry. The acceptor registry then
calls is_collocated() on each acceptor for each endpoint in the mprofile
that has a tag matching the acceptor’s tag.

Your acceptor must return non-zero if the provided endpoint’s object address
exactly matches one of the acceptor’s addresses. The nature of the comparison
depends on the type of the address in the protocol. Assuming your address
type is derived from ACE_Addr (we cover how you can use the ACE classes
to make your implementation of a custom protocol much easier), an appropriate
action is to use the equality operator to perform this function. An example of this use, from the UIOP acceptor, is shown below:

```c
int
TAO_UIOP_Acceptor::is_collocated (const TAO_Endpoint* endpoint)
{
    const TAO_UIOP_Endpoint* endp =
        ACE_dynamic_cast (const TAO_UIOP_Endpoint*, endpoint);

    // Make sure the dynamically cast pointer is valid.
    if (endp == 0)
        return 0;

    // For UNIX Files this is relatively cheap.
    ACE_UNIX_Addr address;
    if (this->base_acceptor_.acceptor ().get_local_addr (address) == -1)
        return 0;
    return endp->object_addr () == address;
}
```

The above function tests equality between the address within the supplied endpoint and the address of the base acceptor. In the UIOP acceptor and most other cases, there is a single address associated with the base acceptor. If your acceptor supports multiple endpoints, like the IIOP acceptor, you should test each base address for equality.

**Pluggable-Acceptor endpoint_count() Function**

**Synopsis**

```c
CORBA::ULong endpoint_count (void)
```

This function returns the number of endpoints used by this acceptor. It is used by the acceptor registry to get an upper bound on the number of profiles that must be included in an IOR. Most implementations can simply return 1.

Some protocols, such as IIOP, may need multiple endpoints to represent acceptors that listen on multiple network interfaces. The IIOP acceptor’s implementation of `open_default()` obtains a list of all interfaces and obtains host identification for each. Thus, if a default endpoint is opened on a machine with multiple network interface cards, clients with network connections to any of the interfaces are able to connect to the server. The IIOP acceptor implementation stores the number of endpoints opened in a data member, then the `endpoint_count()` function simply returns that data member.
16.4 Details of the Pluggable-Protocol Framework

**Pluggable-Acceptor object_key() Function**

**Synopsis**

```c
int object_key (IOP::TaggedProfile& profile,
    TAO_ObjectKey& key)
```

When GIOP 1.1 is used as the messaging protocol, the object key is extracted directly from the GIOP request message header. GIOP 1.2 extends the request message header format to use a union for the target address. This union may take on any one of the following three values:

- A simple object key.
- A profile of type `IOP::TaggedProfile`.
- A reference of type `GIOP::IORAddressingInfo`, which contains an IOR and an index to the selected profile within the IOR.

When the target address in the header is not an object key, the TAO framework relies on the acceptor to extract the key from the supplied profile. If the target address is of type `GIOP::IORAddressingInfo`, the specified profile is extracted from the IOR, and the object key is obtained from that.

A tagged profile is an opaque type, used to communicate vendor and protocol-specific information. In TAO pluggable protocols, the `profile_data` field of the tagged profile is used to store all the information about a particular object on a particular server listening on a particular endpoint for a particular protocol. To obtain efficiency in encoding/decoding this information, this octet sequence is manipulated as a CDR stream. The data stored in the `profile_data` field of the tagged profile object is in the same format as profiles encoded in an IOR. This means that to extract the object key, your `object_key()` function must work its way through the other data in the buffer. The simplest strategy for doing this is to work through the same process as that protocol’s implementation of `TAO_Profile::decode()`, discarding everything up to the object key. This strategy is shown here in the UIOP acceptor implementation:

```c
int
TAO_UIOP_Acceptor::object_key (IOP::TaggedProfile& profile,
    TAO_ObjectKey& object_key)
{
    // Create the decoding stream from the encapsulation in the buffer,
    #if (TAO_NO_COPY_OCTET_SEQUENCES == 1)
        TAO_InputCDR cdr (profile.profile_data.mb());
    #else
        TAO_InputCDR cdr (profile.profile_data.mb());
    #endif
```
Developing Pluggable Protocols

The above function returns -1 on a failure to decode, and 1 on success. This is a departure from the normal behavior for functions of this nature, but it is in keeping with TAO_Profile::decode(), on which this function should be based.

Protocol-Specific parse_options() Function

Synopsis

int TAO_UIOP_Acceptor::parse_options (const char* str)
This function is not part of the base-acceptor class. However, it is important to discuss it here to ensure that the code produced is consistent with other acceptor implementations. In the case of UIOP, there are currently no options that are parsed apart from a deprecated priority value. (Per-endpoint priority is now set via a policy as part of the RT-CORBA specification. See Chapter 9 for more information.)

Through utilities supplied by the protocol factory, the endpoint is broken apart and everything following the option delimiter (“|” in UIOP’s case) is provided as the options string to either `open()` or `open_default()`. The format of this string is arbitrary, but it cannot violate the syntax of the `-ORBListenEndpoints` option. Specifically, it cannot contain spaces, commas, or semicolons.

Options parsed by the IIOP and UIOP acceptors use a “name=value” syntax, where multiple options are separated with the ‘&’ character. The `ACE_CString` class provides some functions that simplify the task of obtaining substrings from strings.

The behavior of `parse_options()` in the TAO-supplied acceptors is as follows:

- Obtain substrings between the ampersands.
- Ensure that the substring contains a name to the left of an equal sign and a value to the right of that equal sign.
- Ensure that the name is valid and that the value makes sense. If any of these conditions are not true, `parse_options()` returns –1. Otherwise, it returns 0 after parsing all options.

16.4.2 The Connector

To support the client’s behavior of actively establishing connections, the pluggable-protocol framework supplies the `TAO_Connector`. Individual connectors are created by the protocol factories during ORB initialization. The connectors are owned by and accessed through the connector registry, which is an element of the ORB core.

The interface for the abstract base-class `TAO_Connector` is shown below. As with the `TAO_Acceptor` class declaration, there are a few function implementations provided for you, such as `tag()`:

```cpp
class TAO_Export TAO_Connector
```
Developing Pluggable Protocols

```cpp
{
public:
    TAO_Connector (CORBA::ULong tag);
    virtual ~TAO_Connector (void);

    CORBA::ULong tag (void) const;
    int make_mprofile (const char *ior,
                      TAO_MProfile &mprofile
                      ACE_ENV_ARG_DECL);
    virtual TAO_Profile *corbaloc_scan (const char *ior,
                                        size_t &len
                                        ACE_ENV_ARG_DECL);
    virtual int open (TAO_ORB_Core *orb_core) = 0;
    virtual int close (void) = 0;
    virtual TAO_Transport* connect (
        TAO::Profile_Transport_Resolver *r,
        TAO_Transport_Descriptor_Interface *desc,
        ACE_Time_Value *timeout
        ACE_ENV_ARG_DECL);
    virtual TAO_Profile *create_profile (TAO_InputCDR& cdr) = 0;
    virtual int check_prefix (const char *endpoint) = 0;
    virtual char object_key_delimiter (void) const = 0;

    protected:
    virtual TAO_Profile *make_profile (ACE_ENV_SINGLE_ARG_DECL) = 0;
    virtual int set_validate_endpoint (TAO_Endpoint *endpoint) = 0;
    virtual int make_connection (TAO_GIOP_Invocation *invocation,
                                  TAO_Transport_Descriptor_Interface *desc,
                                  ACE_Time_Value *timeout) = 0;
    virtual int cancel_svc_handler (  
        TAO_Connection_Handler *svc_handler) = 0;
    virtual int check_connection_closure (  
        TAO_Connection_Handler *connection_handler);
    virtual bool wait_for_connection_completion(
        TAO::Profile_Transport_Resover *r,
        TAO_Transport *&transport,
        ACE_Time_Value *timeout);
    void orb_core (TAO_ORB_Core *orb_core);
    int create_connect_strategy (void);
    TAO_ORB_Core *orb_core (void);

    TAO_Connect_Strategy *active_connect_strategy_;

    private:
    CORBA::ULong tag_;
    TAO_ORB_Core *orb_core_;
};
```
Like all pluggable-protocol elements, the pluggable connectors are identified by an OMG-supplied tag.

The pure virtual functions shown are required of any pluggable connector. These functions are used by the framework to initialize the connector, request new connections, and validate endpoint prefixes.

The protected function `make_profile()` is pure virtual, whereas the public function `make_mprofile()` is not. This is because the behavior of `make_mprofile()` is strictly base-class behavior that uses the protected function `make_profile()` to supply a protocol-specific profile for the list of profiles.

For example, here is the `TAO_UIOP_Connector` class definition:

class TAO_Strategies_Export TAO_UIOP_Connector : public TAO_Connector
{
    public:
        TAO_UIOP_Connector (CORBA::Boolean flag = 0);
        ~TAO_UIOP_Connector (void);
        int open (TAO_ORB_Core *orb_core);
        int close (void);
        TAO_Profile *create_profile (TAO_InputCDR& cdr);
        virtual int check_prefix (const char *endpoint);
        virtual TAO_Profile *corbaloc_scan (const char *str, size_t &len
                ACE_ENV_ARG_DECL);
        virtual char object_key_delimiter (void) const;
        virtual int cancel_svc_handler (TAO_Connection_Handler * svc_handler);

    public:
        typedef TAO_Connect_Concurrency_Strategy<TAO_UIOP_Connection_Handler>
                TAO_UIOP_CONNECT_CONCURRENCY_STRATEGY;
        typedef TAO_Connect_Creation_Strategy<TAO_UIOP_Connection_Handler>
                TAO_UIOP_CONNECT_CREATION_STRATEGY;
        typedef ACE_Connect_Strategy<TAO_UIOP_Connection_Handler,
                ACE_LSOCK_CONNECTOR>
                TAO_UIOP_CONNECT_STRATEGY;
        typedef ACE_Strategy_Connector<TAO_UIOP_Connection_Handler,
                ACE_LSOCK_CONNECTOR>
                TAO_UIOP_BASE_CONNECTOR;

    protected:
        int set_validate_endpoint (TAO_Endpoint *endpoint);
        TAO_Transport *make_connection (TAO_Profile_Transport_Resolver *r,
                TAO_Transport_Descriptor_Interface &desc,
                ACE_Time_Value *timeout = 0);
        virtual TAO_Profile *make_profile (ACE_ENV_SINGLE_ARG_DECL);
private:
    TAO_UIOP_Endpoint *remote_endpoint (TAO_Endpoint *ep);

    TAO_UIOP_CONNECT_STRATEGY connect_strategy_;  // Connect strategy
    TAO_UIOP_BASE_CONNECTOR base_connector_;     // Base connector
    CORBA::Boolean lite_flag_;                   // Lite flag
};

16.4.2.1 The Connector Typedefs and Strategies

The following typedefs are in the concrete acceptor class definition:

typedef TAO_Connect_Concurrency_Strategy<TAO_UIOP_Connection_Handler>  
    TAO_UIOP_CONNECT_CONCURRENCY_STRATEGY;

typedef TAO_Connect_Creation_Strategy<TAO_UIOP_Connection_Handler>  
    TAO_UIOP_CONNECT_CREATION_STRATEGY;

typedef ACE_Connect_Strategy<TAO_UIOP_Connection_Handler, 
    ACE_LSOCK_CONNECTOR>  
    TAO_UIOP_CONNECT_STRATEGY ;

typedef ACE_Strategy_Connector<TAO_UIOP_Connection_Handler, 
    ACE_LSOCK_CONNECTOR>  
    TAO_UIOP_BASE_CONNECTOR;

All protocols currently provided with TAO leverage existing generic code to 
carry out their responsibilities. These typedefs are used to define the types of 
strategy member variables as in:

    TAO_UIOP_CONNECT_STRATEGY connect_strategy_;  
    TAO_UIOP_BASE_CONNECTOR base_connector_;     

Two other strategy objects are instantiated in the connectors `open()` function, 
then passed to the base connector as shown in the following code:

    int TAO_UIOP_Connector::open (TAO_ORB_Core *orb_core) 
    {  
        this->orb_core (orb_core);

        // Create our connect strategy
        if (this->create_connect_strategy () == -1)
            return -1;

        // Our connect creation strategy

Published by Object Computing, Inc. © 2008. All Rights Reserved.
TAO_UIOP_CONNECT_CREATION_STRATEGY *connect_creation_strategy = 0;

ACE_NEW_RETURN (connect_creation_strategy,
    TAO_UIOP_CONNECT_CREATION_STRATEGY
    (orb_core->thr_mgr (),
    orb_core,
    this->lite_flag_),
    -1);

    // Our activation strategy
TAO_UIOP_CONNECT_CONCURRENCY_STRATEGY *concurrency_strategy = 0;

ACE_NEW_RETURN (concurrency_strategy,
    TAO_UIOP_CONNECT_CONCURRENCY_STRATEGY (orb_core),
    -1);

return this->base_connector_.open (this->orb_core ()->reactor (),
    connect_creation_strategy,
    &this->connect_strategy_,
    concurrency_strategy);
Figure 16-2 shows a UML diagram of the connector’s relationship with its strategies. The interclass structure and dynamics mirror that of the acceptor.

### Strategy Connector Template

**Synopsis**

template <class SVC_HANDLER, ACE_PEER_CONNECTOR_1>
class ACE_Strategy_Connector :
    public ACE_Connector <SVC_HANDLER,ACE_PEER_CONNECTOR_2>

This class is a generic and strategized version of the Connector pattern. It is a factory for creating a service handler, connecting into that service handler, and activating that service handler. The type of the service handler is determined at compile time by the template parameter. It performs these actions through the use of various strategy classes that are configured at run time. As with the ACE_Strategy_Acceptor, this class is protocol-independent, but capable of creating and calling protocol-specific code. The two template parameters that are used to define this class are:
1. the type of service handler to be instantiated, connected into, and activated.

2. the particular stream abstraction for the low-level I/O mechanism to use with a certain protocol.

The `ACE_Strategy_Connector` uses a combination of the Template Method pattern and the Strategy pattern to fulfill its role in the Connector pattern. Its base class has a `connect()` function that is called by the upper-level ORB code when the client initiates a connection request. The base class establishes the steps necessary to make this connection request complete and to set up the pluggable-protocol framework for receiving replies from the server. Though the order in which these steps are performed is mandated by the base-class function (as described in the Template Method pattern), the implementation of each of the steps is the responsibility of the derived class `ACE_Strategy_Connector`, which delegates its actions to specific strategy classes to achieve its functionality. The steps prescribed by the `connect()` function are:

- `make_svc_handler()`
- `connect_svc_handler()`
- `activate_svc_handler()`

Below is the key template function with the key steps highlighted that put the protocol-specific connection handler into play:

```cpp
template <class SVC_HANDLER, ACE_PEER_CONNECTOR_1> int
ACE_Connector<SVC_HANDLER, ACE_PEER_CONNECTOR_2>::connect_i
(SVC_HANDLER *sh,  
SVC_HANDLER **sh_copy,  
const ACE_PEER_CONNECTOR_ADDR &remote_addr,  
const ACE_Synch_Options &synch_options,  
const ACE_PEER_CONNECTOR_ADDR &local_addr,  
int reuse_addr,  
int flags,  
int perms) // Save/restore errno.
{
ACE_TRACE ("ACE_Connector<SVC_HANDLER, ACE_PEER_CONNECTOR_2>::connect_i");

    // If the user hasn’t supplied us with a <SVC_HANDLER> we’ll use the
    // factory method to create one. Otherwise, things will remain as
    // they are...
    if (this->make_svc_handler (sh) == -1)
        return -1;
```
ACE_Time_Value *timeout;
int use_reactor = synch_options[ACE_Synch_Options::USE_REACTOR];

if (use_reactor)
    timeout = (ACE_Time_Value *) &ACE_Time_Value::zero;
else
    timeout = (ACE_Time_Value *) synch_options.time_value ();

int result;
if (sh_copy == 0)
    result = this->connect_svc_handler (sh,
                        remote_addr,
                        timeout,
                        local_addr,
                        reuse_addr,
                        flags,
                        perms);
else
    result = this->connect_svc_handler (sh,
                        *sh_copy,
                        remote_addr,
                        timeout,
                        local_addr,
                        reuse_addr,
                        flags,
                        perms);

// Activate immediately if we are connected.
if (result != -1)
    return this->activate_svc_handler (sh);

// Delegate to connection strategy.
if (use_reactor && ACE_OS::last_error () == EWOULDBLOCK)
{
    // If the connection hasn’t completed and we are using
    // non-blocking semantics then register
    // ACE_NonBlocking_Connect_Handler with the ACE_Reactor so that
    // it will call us back when the connection is complete or we
    // timeout, whichever comes first...
    int result;

    if (sh_copy == 0)
        result = this->nonblocking_connect (sh, synch_options);
    else
        result = this->nonblocking_connect (*sh_copy, synch_options);

    // If for some reason the <nonblocking_connect> call failed, then <errno>
    // will be set to the new error. If the call succeeds, however,
    // we need to make sure that <errno> remains set to
    // <EWOULDBLOCK>.
The `connect()` function execution begins in the `ACE_Connector` base class by invoking the above three functions on itself, but their execution is deferred while the derived `ACE_Strategy_Connector` forwards them onto the various strategy objects for actual execution.

These concrete strategy classes are determined by the other `typedefs` and are discussed below.

**Creation-Strategy Template**

**Synopsis**

```cpp
template <class SVC_HANDLER>
class TAO_Connect_Creation_Strategy :
  public ACE_Creation_Strategy<SVC_HANDLER>
```

The creation-strategy class is instantiated with the concrete protocol-specific connection handler as a template parameter. The `make_svc_handler()` function is overridden by the `TAO_Connect_Creation_Strategy` subclass, which instantiates the protocol-specific connection handler, using the proper constructor for the pluggable-protocol framework as follows:

```cpp
template <class SVC_HANDLER> int
TAO_Connect_Creation_Strategy<SVC_HANDLER>::make_svc_handler (SVC_HANDLER*& sh) {
  if (sh == 0)
    ACE_NEW_RETURN (sh,
```
Developing Pluggable Protocols

SVC_HANDLER (this->orb_core_,
this->lite_flag_,
this->arg_),
-1);

// We add to the #REFCOUNT# since the Connector needs this. See
// Connector::make_connection() for details.
sh->add_reference ();

// At this point, the #REFCOUNT# is two.
return 0;
}

Connect-Strategy Template

Synopsis
template <class SVC_HANDLER, ACE_PEER_CONNECTOR_1>
class ACE_Connect_Strategy

The ACE_Connect_Strategy class is instantiated with the concrete
protocol-specific connection handler and stream connector as template
parameters. The connect_svc_handler() function of this class simply
establishes the new connection into the protocol-specific connection handler
as shown below:

template <class SVC_HANDLER, ACE_PEER_CONNECTOR_1> int
ACE_Connect_Strategy<SVC_HANDLER, ACE_PEER_CONNECTOR_2>::connect_svc_handler
(SVC_HANDLER*& sh,
const ACE_PEER_CONNECTOR_ADDR& remote_addr,
ACE_Time_Value* timeout,
const ACE_PEER_CONNECTOR_ADDR& local_addr,
int reuse_addr,
int flags,
int perms)
{
    ACE_TRACE ("ACE_Connect_Strategy<SVC_HANDLER,
ACE_PEER_CONNECTOR_2>::connect_svc_handler");

    return this->connector_.connect (sh->peer (),
remote_addr,
timeout,
local_addr,
reuse_addr,
flags,
perms);
}
Concurrency-Strategy Template

Synopsis

```cpp
template <class SVC_HANDLER>
class TAO_Connect_Concurrency_Strategy :
    public ACE_Concurrency_Strategy<SVC_HANDLER>
```

The concurrency-strategy class is instantiated with the concrete protocol-specific connection handler as a template parameter, and overrides the `activate_svc_handler()` function. The implementation shown below calls the base-class functionality and the `activate_svc_handler()`, which registers the connection handler with the reactor:

```cpp
template <class SVC_HANDLER> int
TAO_Concurrency_Strategy<SVC_HANDLER>::activate_svc_handler (SVC_HANDLER *sh, 
    void *arg)
{
    sh->transport ()->opened_as (TAO::TAO_SERVER_ROLE);

    // Indicate that this transport was opened in the server role
    if (TAO_debug_level > 6)
        ACE_DEBUG ((LM_DEBUG, 
            "TAO (%P|%t) - Concurrency_Strategy::activate_svc_handler, " 
            "opened as TAO_SERVER_ROLE\n"));

    // Here the service handler has been created and the new connection 
    // has been accepted.  #REFCOUNT# is one at this point.
    if (this->ACE_Concurrency_Strategy<SVC_HANDLER>::activate_svc_handler (sh, 
        arg) == -1)
        return -1;

    // The service handler has been activated. Now cache the handler.
    if (sh->add_transport_to_cache () == -1)
    {
        // Adding to the cache fails, close the handler.
        sh->close ();

        // #REFCOUNT# is zero at this point.
        if (TAO_debug_level > 0)
        {
            ACE_ERROR ((LM_ERROR, 
                ACE_TEXT ("TAO (%P|%t) - " 
                Concurrency_Strategy::activate_svc_handler, "
            
                ACE_TEXT ("could not add the handler to cache \n")));
        }

        return -1;
    }
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
Developing Pluggable Protocols

TAO_Server_Strategy_Factory *f =
    this->orb_core_->server_factory ();

int result = 0;

// Do we need to create threads?
if (f->activate_server_connections ())
{
    // Thread-per-connection concurrency model
    TAO_Thread_Per_Connection_Handler *tpch = 0;

    ACE_NEW_RETURN (tpch,
        TAO_Thread_Per_Connection_Handler (sh,
            this->orb_core_),
        -1);

    result =
        tpch->activate (f->server_connection_thread_flags (),
            f->server_connection_thread_count ());
}
else
{
    // Otherwise, it is the reactive concurrency model. We may want
    // to register ourselves with the reactor. Call the register
    // handler on the transport.
    result =
        sh->transport ()->register_handler ();
}

if (result != -1)
{
    // Activation/registration successful: the handler has been
    // registered with either the Reactor or the
    // Thread-per-Connection_Handler, and the Transport Cache.
    // #REFCOUNT# is three at this point.

    // We can let go of our reference.
    sh->transport ()->remove_reference ();
}
else
{
    // Activation/registration failure. #REFCOUNT# is two at this
    // point.

    // Remove from cache.
    sh->transport ()->purge_entry ();

    // #REFCOUNT# is one at this point.
16.4 Details of the Pluggable-Protocol Framework

// Close handler.
sh->close ();

// #REFCOUNT# is zero at this point.

if (TAO_debug_level > 0)
{
    const ACE_TCHAR *error = 0;
    if (f->activate_server_connections ())
        error = ACE_TEXT("could not activate new connection");
    else
        error = ACE_TEXT("could not register new connection in the reactor");

    ACE_ERROR ((LM_ERROR,
        "TAO (%P|%t) - Concurrency_Strategy::activate_svc_handler, 
        "%s\n", error));
}

return -1;
}

// Success: #REFCOUNT# is two at this point.
return result;

16.4.2.2 Member Functions in Detail

Each of the functions declared in the interface to the connector class are described below.

Pluggable-Connector Constructor

The connector is constructed by the protocol factory. Therefore the signature of your constructor is entirely up to you. It must call the TAO_Connector constructor, supplying it with the tag value reserved for your protocol. Apart from that, your constructor should only initialize state. Any initialization related to connection establishment, such as address assignment, should be deferred until open() is called. The following is an example of the TAO_UIOP_Connector constructor:

TAO_UIOP_Connector::TAO_UIOP_Connector (CORBA::Boolean flag)
: TAO_Connector (TAO_TAG_UIOP_PROFILE),
  connect_strategy_ (),
  base_connector_ (),
  lite_flag_ (flag)
{
We initialize the base constructor with TAO_TAG_UIOP_PROFILE, the unique identifier for UIOP profiles. The base_connector() member is the protocol-specific connector (as a result of the protocol-specific template parameterization of the ACE_Strategy_Connector class—see the typedefs in the TAO_UIOP_Connector class definition). The base-connector initialization is discussed further in the description of the open() function.

The connector passes the flag argument later to strategies that it instantiates. In the case of UIOP and IIOP, the acceptor and connector are used in two forms: one that supports the standard CORBA messaging protocol GIOP; and the other that supports a TAO-specific optimization of GIOP, known as GIOP “lite.” There are different protocol factories used to differentiate these two messaging protocols, but they use the same acceptor, connector, etc.

**Pluggable-Connector Destructor**

The base-class destructor serves no purpose other than to define the virtual function so that proper clean up occurs when deletion of a derived object occurs through the use of a base-class pointer. If your connector owns dynamically-allocated objects, you need to supply a destructor to clean up.

**Pluggable-Connector open() Function**

**Synopsis**

```c
int open (TAO_ORB_Core* orb_core)
```

Opening a connector is somewhat less complex than opening an acceptor, because an acceptor must initialize an endpoint right away in its open() function. For a connector, the endpoint is not initialized until it is actually called upon to establish a connection to a server. As we saw in the example at the beginning of this chapter, a pluggable connector is opened by the connector registry when the ORB is initialized.

The example below shows how the UIOP connector is opened. The open() function initializes the ORB-core reference and the base connector.

The ACE_Strategy_Connector takes a series of strategy objects that customize the connector behavior. This code shows how the strategy objects are initialized:

```c
int
TAO_UIOP_Connector::open (TAO_ORB_Core* orb_core)
{ 
```
16.4 Details of the Pluggable-Protocol Framework

```c
this->orb_core (orb_core);

if (this->create_connect_strategy () != 0)
    return -1;

// Our connect creation strategy
TAO_UIOP_CONNECT_CREATION_STRATEGY* connect_creation_strategy = 0;
ACE_NEW_RETURN (connect_creation_strategy,
    TAO_UIOP_CONNECT_CREATION_STRATEGY
    (orb_core->thr_mgr (),
     orb_core,
     this->lite_flag_),
    -1);

/// Our activation strategy
TAO_UIOP_CONNECT_CONCURRENCY_STRATEGY* concurrency_strategy = 0;
ACE_NEW_RETURN (concurrency_strategy,
    TAO_UIOP_CONNECT_CONCURRENCY_STRATEGY (orb_core),
    -1);

return this->base_connector_.open (this->orb_core ()->reactor (),
    connect_creation_strategy,
    &this->connect_strategy_,
    concurrency_strategy);
}
```

After the required strategies are allocated, these are used to open the base connector.

### Pluggable-Connector close() Function

**Synopsis**

```c
int close (void)
```

The close() function signals the end of the connector life cycle. Any resources opened during open() should be closed, such as the base connector. To maintain symmetry in your code, any resources allocated during open() should be reclaimed during close(). If you are modeling your code on the TAO-supplied connectors, and therefore not implementing the destructor, then you must carefully reclaim dynamic memory during close().

Here is the close() implementation for the UIOP connector:

```c
int
TAO_UIOP_Connector::close (void)
{
```
Developing Pluggable Protocols

// Zap the creation strategy that we created earlier.
delete this->base_connector_.creation_strategy();
delete this->base_connector_.concurrency_strategy();

return this->base_connector_.close();
}

Pluggable-Connector corbaloc_scan() Function

Synopsis

TAO_Profile * corbaloc_scan (const char *str, size_t &len ACE_ENV_ARG_DECL)

This method is used to help parse corbaloc style object references. It examines
the supplied string to locate the end of a protocol-specific endpoint
description. The corbaloc_scan() method is required to return a pointer to
a newly constructed TAO_Profile object if the protocol identifier tag is
recognized. The method searches for a separator character to identify the end
of the address specification. An implementation of corbaloc_scan() is
provided in the base TAO_Transport_Connector class. This base
implementation of corbaloc_scan() is usually sufficient, as it uses the
comma (',') and slash ('/') characters for detecting the end of an address.
UIOP is distinct in that its endpoint is a filesystem path, and both comma and
slash are valid characters in a path. For that reason, UIOP based corbaloc
references use the pipe ('|') as a prefix for the separator.

Here is the corbaloc_scan() implementation for the UIOP connector.
Notice at the end, the length argument is updated. This allows the caller to do
further parsing on the corbaloc string.

TAO_Profile *
TAO_UIOP_Connector::corbaloc_scan (const char *str, size_t &len
ACE_ENV_ARG_DECL)
{
if (this->check_prefix (str) != 0)
    return 0;

const char *separator = ACE_OS::strchr (str,'|');
if (separator == 0)
{
    if (TAO_debug_level)
        ACE_DEBUG ((LM_DEBUG,
            "(%P|%t) TAO_UIOP_CONNECTOR::corbaloc_scan error: "
            "explicit terminating character '|' is missing from <%s>",
            str));

    return 0;
}
16.4 Details of the Pluggable-Protocol Framework

```c
if (*\(separator+1\) != ',' && *(separator+1) != '/')
{
    if (TAO_debug_level)
    
    ACE_DEBUG ((LM_DEBUG,
                  "(%P|%t) TAO_UIOP_CONNECTOR::corbaloc_scan warning: ",
                  "terminating character '|' should be followed by a ',' ",
                  "or a '/' in <\%s>",
                  str));
}

len = separator - str) + 1;
return this->make_profile (ACE_ENV_SINGLE_ARG_PARAMETER);
```

Pluggable-Connector connect() Function

**Synopsis**

```c
TAO_Transport * make_connection (TAO::Profile_Transport_Resolver *r,
    TAO_Transport_Descriptor_Interface &desc,
    ACE_Time_Value *max_wait_time)
```

The `make_connection()` function is the “business” function of a pluggable connector. This function is called when the ORB cannot find a usable connection for the endpoint. There are two possible outcomes of a call to `make_connection()`: either a successful connection is established, with the transport referring to a valid object; or an error occurs.

The expected behavior of `make_connection()` involves a few fundamental steps. The following code fragments illustrate these steps by examining the `make_connection()` function of the `TAO_UIOP_Connector`:

```c
TAO_Transport *
TAO_UIOP_Connector::make_connection (TAO::Profile_Transport_Resolver *r,
    TAO_Transport_Descriptor_Interface &desc,
    ACE_Time_Value *max_wait_time)
```

```c
if (TAO_debug_level > 0)
    ACE_DEBUG ((LM_DEBUG,
                  ACE_TEXT ("TAO (%P|%t) UIUP_Connector::make_connection, "))
                  ACE_TEXT ("looking for UIOP connection.\n"));

TAO_UIOP_Endpoint *uiop_endpoint =
this->remote_endpoint (desc.endpoint());
```

The first step is to obtain the remote address from the endpoint in the provided transport descriptor. The TAO-supplied connectors perform extensive error/sanity checking on the profile and the address contained within. If
remote_endpoint returns NULL, then the supplied endpoint was not a match for this protocol.

    if (uiop_endpoint == 0)
        return -1;

The next step is to attempt to connect to the peer. Depending on the configuration, this may be a blocking call with our without a timeout, or an asynchronous call with or without timeout. The meaning of asynchronous in this case is that the caller returns after registering the pending connection with the reactor.

    const ACE_UNIX_Addr &remote_address =
        uiop_endpoint->object_addr ();

    if (TAO_debug_level > 2)
        ACE_DEBUG ((LM_DEBUG,
            ACE_TEXT("(%P|%t) UIOP_Connector::connect ")
            ACE_TEXT("making a new connection \n"));

    ACE_Synch_Options synch_options;

    this->active_connect_strategy_->synch_options (max_wait_time,
                                                synch_options);

    ACE_Time_Value tmp_zero (ACE_Time_Value::zero);
    if (!r->blocked ())
        { synch_options.timeout (ACE_Time_Value::zero);
          max_wait_time = &tmp_zero;
        }

    TAO_UIOP_Connection_Handler *svc_handler = 0;

    int result =
        this->base_connector_.connect (svc_handler,
                                        remote_address,
                                        synch_options);

If the connection options are not set to asynchronous, the call may block, with or without timeout.

    ACE_Event_Handler_var svc_handler_auto_ptr (svc_handler);

    TAO_Transport *transport =
        svc_handler->transport ();
if (result == -1)
{
    // No immediate result, wait for completion
    if (errno == EWOULDBLOCK)
    {
        // Try to wait until connection completion. Incase we block, then we
        // get a connected transport or not. In case of non block we get
        // a connected or not connected transport
        if (!this->wait_for_connection_completion (r,
            transport,
            max_wait_time))
        {
            if (TAO_debug_level > 2)
                ACE_ERROR ((LM_ERROR, "TAO (%P|%t) - UIOP_Connector::" "make_connection, " "wait for completion failed\n");
        }
    }
    else
    {
        // Transport is not usable
        transport = 0;
    }
}

Finally we check for error conditions. If there is not a problem, we register our
handler and transport.

// In case of errors transport is zero
if (transport == 0)
{
    // Give users a clue to the problem.
    if (TAO_debug_level > 3)
        ACE_DEBUG ((LM_ERROR,
            "TAO (%P|%t) - UIOP_Connector::make_connection, " "connection to <%s> failed (%p)\n",
            uiop_endpoint->rendezvous_point ()
            , ACE_TEXT("errno")))

    return 0;
}

// At this point, the connection has been successfully created and
// connected or not connected, but we have a connection.
if (TAO_debug_level > 2)
    ACE_DEBUG ((LM_DEBUG,
        "TAO (%P|%t) - UIOP_Connector::make_connection, " "new %s connection to <%s> on Transport[%d]\n",
        ACE_TEXT("errno")))


transport->is_connected() ? "connected" : "not connected",
uiop_endpoint->rendezvous_point (),
svc_handler->peer ().get_handle ());

// Add the handler to Cache
int retval =
  this->orb_core ()->lane_resources ()->transport_cache ().
    cache_transport (&desc, transport);

// Failure in adding to cache.
if (retval != 0)
{
  // Close the handler.
  svc_handler->close ();

  if (TAO_debug_level > 0)
  {
    ACE_ERROR ((LM_ERROR,
      ACE_TEXT("TAO (%P|%t) UIOP_Connector::make_connection, ")
      ACE_TEXT("could not add the new connection to Cache \n")));
  }

  return 0;
}

if (transport->is_connected () &&
    transport->wait_strategy ()->register_handler () != 0)
{
  // Registration failures.

  // Purge from the connection cache, if we are not in the cache, this
  // just does nothing.
  transport->purge_entry ();

  // Close the handler.
  transport->close_connection ();

  if (TAO_debug_level > 0)
    ACE_ERROR ((LM_ERROR,
      "TAO (%P|%t) - UIOP_Connector [%d]::make_connection, "
      "could not register the transport "
      "in the reactor.\n",
      transport->id ()))

  return 0;
}

return transport;
16.4 Details of the Pluggable-Protocol Framework

Pluggable-Connector create_profile() Function

Synopsis

TAO_Profile* create_profile (TAO_InputCDR& cdr)

Another important task of the connector is to assist in the decomposition of object references obtained as the result of operation invocations. Object references contain profiles for all of the protocols supported by the server in which the object is currently active. Profiles are identified in the CDR by the protocol identifier tag. This is the same tag that is passed to the constructor of TAO_Connector (and TAO_Acceptor). The tag is used to identify the appropriate connector for performing the decoding. A default profile type, TAO_Unknown_Profile, is used to obtain the profile information from the CDR stream if the client does not have a connector for the type of protocol desired.

The connectors are chosen to perform this activity because of the structure of the framework. CDR streams are used to encapsulate the data transmitted between clients and servers as part of invocation requests or replies. TAO relies on C++ insertion and extraction operators to encode and decode CDR streams. When object references are extracted, the task of obtaining profiles is delegated to the connector registry obtained from the ORB core. The connector registry encapsulates the knowledge of which protocols are supported by the client. And since the connector registry holds a mapping between protocol identifier tags and connectors, it is natural for the connectors to be factories for the profile objects.

The actual creation of profiles is a simple operation since profiles know how to decode themselves from the input stream. It is only necessary that your connector create a profile object of the appropriate type, then allow it to decode its contents from the CDR stream. This example shows how the TAO_UIOP_Connector creates a new profile object:

TAO_Profile*
TAO_UIOP_Connector::create_profile (TAO_InputCDR& cdr)
{
    TAO_Profile* pfile;
    ACE_NEW_RETURN (pfile,
        TAO_UIOP_Profile (this->orb_core ()),
        0);

    int r = pfile->decode (cdr);
    if (r == -1) {
        pfile->_decr_refcnt ();
        pfile = 0;
    }
First, a profile object is created on the heap, and a base TAO_Profile pointer is initialized to point to the profile object (because TAO_Profile objects are polymorphic, this is appropriate). Then the profile decodes the CDR stream contents, returning 0 upon success, or -1 upon failure.

If a failure occurs, the profile object is not directly deleted. Rather, its reference count is decremented. When the count becomes zero, the profile deletes itself. So even though the profile is an ordinary object at this point, it is wiser to treat it like any other reference-counted object, and allow it to delete itself as needed. Of course, the pointer we initialized earlier must be reset to NULL, as it is the return value of the function.

**Pluggable-Connect check_prefix() Function**

**Synopsis**

```c
int check_prefix (const char* endpoint)
```

This is a boolean function that is used to determine whether the connector is able to establish connections to the supplied endpoint. Always ensure that all acceptable prefixes are checked (i.e., the TAO_UIOP_Connector checks for uiop and uioploc). Endpoints have the form prefix://... and are case insensitive, so your code should check that the provided endpoint string starts with any one of the allowed prefixes, and ends with a colon, or “://”.

Here is the check_prefix implementation for TAO_UIOP_Connector:

```c
int TAO_UIOP_Connector::check_prefix (const char* endpoint)
{
    // Check for a valid string
    if (!endpoint || !*endpoint)
        return -1; // Failure

    const char* protocol[] = { "uiop", "uioploc" };

    size_t slot = ACE_OS::strchr (endpoint, ':') - endpoint;
    size_t len0 = ACE_OS::strlen (protocol[0]);
    size_t len1 = ACE_OS::strlen (protocol[1]);

    // Check for the proper prefix in the IOR. If the proper prefix
    // isn't in the IOR then it is not an IOR we can use.
16.4 Details of the Pluggable-Protocol Framework

if (slot == len0
    && ACE_OS::strncasecmp (endpoint,
                           protocol[0],
                           len0) == 0)
    return 0;
else if (slot == len1
    && ACE_OS::strncasecmp (endpoint,
                           protocol[1],
                           len1) == 0)
    return 0;
return -1;
// Failure: not an UIOP IOR DO NOT throw an exception here.
}

Although this function is boolean in nature, it follows the ACE (and common system function call) convention of returning 0 for success and –1 for failure.

**Pluggable-Connector object_key_delimiter() Function**

**Synopsis**

```c
char object_key_delimiter (void)
```

This is a pass-through function that returns the object-key delimiter value used by the associated profile class. This convenience function is provided so that the connector registry does not have to determine the specific type of profile it should reference, when attempting to interpret a text-based object reference. The following example shows how to obtain the object-key delimiter from the static data member of the related profile class:

```c
char
TAO_UIOP_Connector::object_key_delimiter (void) const
{
    return TAO_UIOP_Profile::object_key_delimiter_;
}
```

**Pluggable-Connector make_profile() Function**

**Synopsis**

```c
TAO_Profile* make_profile (ACE_ENV_SINGLE_ARG_DECL)
```

This is a factory function used to generate a new protocol-specific profile object. The `make_profile()` function is part of the implementation of `CORBA::ORB::string_to_object()` and is used when the provided IOR string is in URL format.
An ordinary IOR string encodes the contents of an octet sequence (that may be interpreted as a CDR stream) representing the object reference information. A URL-formatted object reference contains this same information in a somewhat more human-readable form. Therefore, during stream parsing, the `CORBA::ORB` class calls on the connector registry to build a multi-profile object out of the profiles contained in the string. The registry extracts the semicolon-separated profile strings, then passes those to the appropriate connector as determined by a call to the connector’s `check_prefix()` function. The connector has a function `make_mprofile()` that further separates out multiple comma-separated endpoints from the profile string and calls `make_profile()` for each endpoint found.

The `TAO_Profile` class for your protocol must provide a default constructor. Therefore, your `make_profile()` does nothing more than create a new instance of the profile object. Your connector should also be prepared to throw an exception if the memory for the profile could not be allocated. An example of `make_profile()` is shown below:

```c++
TAO_Profile*
TAO_UIOP_Connector::make_profile (ACE_ENV_SINGLE_ARG_DECL)
{
    TAO_Profile* profile = 0;
    ACE_NEW_THROW_EX (profile,
        TAO_UIOP_Profile (this->orb_core ()),
        CORBA::NO_MEMORY (
            CORBA::SystemException::tao_minor_code (CORBA_DEFAULT_MINOR_CODE, ENOMEM),
            CORBA::COMPLETED_NO));

    ACE_CHECK_RETURN (0);

    return profile;
}
```

### 16.4.3 The Connection Handler

The connection handler represents the third part of the streaming-protocol triad. Once the acceptors and connectors have established a connection, it is the service handler, in this case a connection handler, in conjunction with the transport that uses the connection to pass data between peers.

Your connection handler can inherit from both `TAO_Connection_Handler` and the `ACE_Svc_Handler<PEER_STREAM, SYNCH>` template class. The
base type, `ACE_Svc_Handler`, is capable of performing many roles. Figure 16-3 shows how this class inherits from `ACE_Task<SYNCH>`, which inherits from `ACE_Service_Object`, which inherits from `ACE_Event_Handler`. The interface to `ACE_Svc_Handler<PEER_STREAM, SYNCH>` is rather thin, but important. It is used here because of its ability to abstract the data-stream interface by use of the peer-stream template argument. When using the abstract peer stream, connection handlers behave primarily like event handlers, dealing with such things as data availability and time-outs, but delegating the actual sending and receiving of data to other classes.

![Figure 16-3 The Connection-Handler Class Diagram](image-url)
The connection handler is created by either the acceptor or the connector when a connection is established. It is responsible for creating an instance of the concrete protocol-specific derived class of the TAO_Transport abstract base class, which is used to handle inter-ORB-protocol messaging. The connection handler works closely with the transport to provide ORB-to-ORB messaging.

The following interface presentation is different from the others in this chapter in that the base class (ACE_Svc_Handler<>) has a large interface, much of which is not relevant to its use as a connection handler in this framework.

TAO also supplies a TAO_Connection_Handler base class that abstracts out many of the common behaviors of all connection handlers. Using this class as a second base class simplifies your implementation of the connection handler. All of TAO’s protocol implementations derive from this class. Notice that all TAO’s protocol implementations rely, in some fashion, on sockets, thus there is a socket-related protected member function in the TAO_Connection_Handler class. If your protocol does not use sockets in any way, then you can either create your own base class or ignore the socket-related function and use the existing base class.

This interface example is based on the connection-handler definition supporting the UIOP protocol. From UIOP_Transport.h:

typedef ACE_Svc_Handler<ACE_LSOCK_STREAM, ACE_NULL_SYNCH> TAO_UIOP_SVC_HANDLER;

The use of a typedef here removes the need to retype a long template name repeatedly. This typedef shows that UIOP uses ACE_LSOCK_Stream for its interprocess communication. Since a single connection handler is used in the context of a single thread, the ACE_NULL_SYNCH locking strategy is used:

class TAO_Strategies_Export TAO_UIOP_Connection_Handler : public

    TAO_UIOP_SVC_HANDLER,
    public TAO_Connection_Handler
{

public:
    TAO_UIOP_Connection_Handler (ACE_Thread_Manager* t = 0);
    TAO_UIOP_Connection_Handler (TAO_ORB_Core *orb_core,
        CORBA::Boolean flag);
    ~TAO_UIOP_Connection_Handler (void);
    virtual int open_handler (void *);
    int close (u_long = 0);
    virtual int open (void *);
    virtual int resume_handler (void);
16.4 Details of the Pluggable-Protocol Framework

```c
virtual int close_connection (void);
virtual int handle_input (ACE_HANDLE);
virtual int handle_output (ACE_HANDLE);
virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask);
virtual int handle_timeout (const ACE_Time_Value &current_time,
                          const void *act = 0);

int add_transport_to_cache (void);
```

protected:
```
  virtual int release_os_resources (void);
};
```

The `handle_close()`, `handle_input()`, and `open()` functions are derived from the `ACE_Event_Handler` and are used to integrate with the reactor. Server-side connection handlers may be either reactive or threaded, depending on the concurrency policy of the acceptor. Server connection handlers are reference counted to allow for the possibility of nested calls. The `add_transport_to_cache()` function is used by the concurrency connection strategy.

The remaining discussion of connection handlers in this section assumes that you are using the `TAO_Connection_Handler` base class. Implementation of a connection handler without the use of the code in this base class is much more involved and beyond the scope of this chapter.

16.4.3.1 The Member Functions in Detail

Pluggable-Connection-Handler Constructor
This constructor is used by the acceptor and connector creation strategy to create a connection handler for each connection. The `orb_core` is passed to the `TAO_Connection_Handler` constructor and the thread manager (from the ORB core) is passed to the service-handler constructor. The `arg` parameter contains protocol-specific properties. The `flag` parameter specifies whether the GIOP-lite messaging protocol should be used.

The UIOP connection-handler’s constructor is shown below:

```c
TAO_UIOP_Connection_Handler::TAO_UIOP_Connection_Handler (  
  TAO_ORB_Core* orb_core,
  CORBA::Boolean flag,
  void* arg)  
  : TAO_UIOP_SVC_HANDLER (orb_core->thr_mgr (), 0, 0),
```
Developing Pluggable Protocols

```
TAO_Connection_Handler (orb_core)
{
    TAO_UIOP_Transport* specific_transport = 0;
    ACE_NEW(specific_transport,
            TAO_UIOP_Transport(this, orb_core, flag));

    // store this pointer (indirectly increment ref count)
    this->transport(specific_transport);
}
```

This constructor initializes the base classes. In the body of the constructor, the UIOP transport object is created on the heap, then passed to the TAO_Connection_Handler base class via the transport() function.

**Pluggable-Connection-Handler open() Function**

**Synopsis**

```
int open (void*)
```

This function is called after the handler is completely connected. It can be used to initialize any protocol-specific properties of the connection. Calling post_open on the transport sets the handle as the id of the transport.

The UIOP connection-handler’s implementation sets the send and receive buffer sizes (as set in the ORB parameters), makes the connection non-blocking, and invokes the transport post_open as shown below:

```
int TAO_UIOP_Connection_Handler::open (void*)
{
    TAO_UIOP_Protocol_Properties protocol_properties;

    protocol_properties.send_buffer_size_ =
        this->orb_core ()->orb_params ()->sock_sndbuf_size ();
    protocol_properties.recv_buffer_size_ =
        this->orb_core ()->orb_params ()->sock_rcvbuf_size ();

    TAO_Protocols_Hooks *tph =
        this->orb_core ()->get_protocols_hooks ();

    bool client =
        this->transport ()->opened_as () == TAO::TAO_CLIENT_ROLE;;

    ACE_DECLARE_NEW_CORBA_ENV;

    ACE_TRY
    {
        if (client)
```
16.4 Details of the Pluggable-Protocol Framework

{  
thp->client_protocol_properties_at_orb_level  
   (protocol_properties  
     ACE_ENV_ARG_PARAMETER);  
     ACE_TRY_CHECK;  
   )  
else  
{  
thp->server_protocol_properties_at_orb_level (  
   protocol_properties  
     ACE_ENV_ARG_PARAMETER);  
     ACE_TRY_CHECK;  
   }
}

ACE_CATCHANY  
{  
   return -1;
}

ACE_ENDTRY;

ACE_CHECK_RETURN (-1);

if (this->set_socket_option (this->peer (),  
   protocol_properties.send_buffer_size_,  
   protocol_properties.recv_buffer_size_) == -1)  
   return -1;

if (this->transport ()->wait_strategy ()->non_blocking ()) {  
   if (this->peer ().enable (ACE_NONBLOCK) == -1)  
      return -1;
}

// Called by the <Strategy_Acceptor> when the handler is completely  
// connected.
ACE_UNIX_Addr addr;

if (this->peer ().get_remote_addr (addr) == -1)  
   return -1;

if (TAO_debug_level > 0)  
   ACE_DEBUG ((LM_DEBUG,  
      ACE_TEXT ("TAO (%P|%t) UIOP connection to server ")  
      ACE_TEXT ("<%s> on %d\n"),  
      addr.get_path_name (), this->peer ().get_handle ());

if (!this->transport ()->post_open ((size_t) this->get_handle ()))  
   return -1;

this->state_changed (TAO_LF_Event::LFS_SUCCESS);

return 0;
Pluggable-Connection-Handler resume_handler() Function

Synopsis  int resume_handler (void)

Called to figure out whether the handler needs to be resumed by the reactor or the application can take care of it. The default value of 0 is returned to allow the reactor to take care of resumption of the handler. The application can return a value more than zero and decide to resume the handler itself. Here is the UIOP implementation:

```c
int TAO_UIOP_Connection_Handler::resume_handler (void)
{
    return ACE_Event_Handler::ACE_APPLICATION_RESUMES_HANDLER;
}
```

Pluggable-Connection-Handler close_connection() Function

Synopsis  int close_connection (void)

Used to actively close underlying connections. Generally delegates the call to the base class close_connection_eh which removes the specific handler from the reactor list and purges the transport entry. Here is the UIOP implementation:

```c
int TAO_UIOP_Connection_Handler::close_connection (void)
{
    return this->close_connection_eh (this);
}
```

Pluggable-Connection-Handler handle_output() Function

Synopsis  int handle_output (ACE_HANDLE handle)

Called when output events are possible (e.g., when flow control abates or non-blocking connection completes). Here is the UIOP implementation:

```c
int TAO_UIOP_Connection_Handler::handle_output (ACE_HANDLE handle)
{
    const int result =
        this->handle_output_eh (handle, this);
```
if (result == -1)
{
    this->close_connection ();
    return 0;
}

return result;

Pluggable-Connection-Handler handle_input() Function

Synopsis

int handle_input (ACE_HANDLE handle)

This is the callback function used by the reactor (when a reactor is being used) to notify your class that data is available on the input stream. If your server uses the thread-per-connection concurrency strategy, the connection handler is run in its own thread and must detect input on its own. The connection-handler class quickly delegates the actual handling of the input to the protocol-specific transport class through the use of the base-class handle_input_eh which in turn calls back into the transport. Here are the UIOP connection-handler’s implementations:

int
TAO_UIOP_Connection_Handler::handle_input (ACE_HANDLE h)
{
    return this->handle_input_eh (h, this);
}

Pluggable-Connection-Handler handle_timeout() Function

Synopsis

int handle_timeout (const ACE_Time_Value &, const void *)

This is used by the reactor to indicate that the connection has timed out. The UIOP implementation is shown below:

int
TAO_UIOP_Connection_Handler::handle_timeout (const ACE_Time_Value &,
                                           const void *)
{
    return this->close ();
}
Pluggable-Connection-Handler add_transport_to_cache() Function

Synopsis

int TAO_UIOP_Connection_Handler::add_transport_to_cache (void)

This function is not defined in a base class, however its implementation is mandatory because it is used by the concurrency strategy template to add the transport associated with this connection to the ORB’s cache. Here is the UIOP implementation:

```c
int TAO_UIOP_Connection_Handler::add_transport_to_cache (void)
{
    ACE_UNIX_Addr addr;

    // Get the peername.
    if (this->peer ().get_remote_addr (addr) == -1)
        return -1;

    // Construct an UIOP_Endpoint object
    TAO_UIOP_Endpoint endpoint (addr);

    // Construct a property object
    TAO_Base_Transport_Property prop (&endpoint);

    TAO_Transport_Cache_Manager &cache=
        this->orb_core ()->lane_resources ().transport_cache ();

    // Add the handler to the Cache
    return cache.cache_idle_transport (&prop,
        this->transport ());
}
```

The above implementation creates an endpoint corresponding to the remote address that has been connected to, then uses that endpoint to construct a TAO_Base_Transport_Property object which is used as the key to this entry in the cache.

Pluggable-Connection-Handler resume_handler() Function

Synopsis

int resume_handler (void)

This function is needed by the lower-level reactor code. All currently-implemented TAO protocols implement this the same way, returning the value of ACE_Event_Handler::ACE_APPLICATION_RESUMES_HANDLER. Your implementation should follow the example of the UIOP code shown below:
16.4 Details of the Pluggable-Protocol Framework

```c
int TAO_UIOP_Connection_Handler::resume_handler (void)
{
    return ACE_Event_Handler::ACE_APPLICATION_RESUMES_HANDLER;
}
```

### 16.4.4 The Endpoint

The endpoint represents addressing information for a single acceptor endpoint and contains everything necessary for a client to contact a server. Each profile contains one or more endpoints. Your protocol-specific endpoint class is defined by publicly deriving from the `TAO_Endpoint` base class:

```c
class TAO_Export TAO_Endpoint
{
public:
    TAO_Endpoint (CORBA::ULong tag,
                  CORBA::Short priority = TAO_INVALID_PRIORITY);
    virtual ~TAO_Endpoint (void);

    CORBA::ULong tag (void) const;
    void priority (CORBA::Short priority);
    CORBA::Short priority (void) const;
    virtual CORBA::Boolean is_equivalent (const TAO_Endpoint* other_endpoint) = 0;
    virtual TAO_Endpoint* next (void) = 0;
    virtual int addr_to_string (char* buffer, size_t length) = 0;
    virtual TAO_Endpoint* duplicate (void) = 0;
    virtual CORBA::ULong hash (void) = 0;

protected:
    TAO_SYNCH_MUTEX addr_lookup_lock_;
    CORBA::ULong hash_val_;
    CORBA::ULong tag_;
    CORBA::Short priority_;

private:
    ACE_UNIMPLEMENTED_FUNC (TAO_Endpoint (const TAO_Endpoint&))
    ACE_UNIMPLEMENTED_FUNC (void operator= (const TAO_Endpoint&))
};
```

This base class contains the `priority_` and `tag_` data members that all endpoints require. The `tag_` is set via the `TAO_Endpoint` constructor. Your derived class needs to set the `priority_` data member using the supplied modifier function. Derived endpoint classes need to define the various pure virtual functions as described below. These functions allow the rest of the
pluggable-protocol framework to compare, duplicate, and organize the various endpoints.

**Pluggable Endpoint Constructor**
The endpoint is constructed by the acceptor as part of profile creation and needs to keep a tag that specifies its protocol. The CORBA priority is the priority of the acceptor that the endpoint is representing. This is part of TAO’s “prioritized endpoints” architecture, and is currently used for RT CORBA only. The TAO protocol implementations all represent the endpoint internally, using the ACE_*_Addr classes, and specify the address via the constructor. For example, the UIOP endpoint constructor takes an ACE_UNIX.Addr object as an argument as shown below:

```c++
TAO_UIOP_Endpoint::TAO_UIOP_Endpoint (const ACE_UNIX.Addr& addr,
                                         CORBA::Short priority)
    : TAO_Endpoint (TAO_TAG_UIOP_PROFILE, priority),
      object_addr_ (addr),
      next_ (0)
{
}
```

Note how the appropriate protocol tag is passed to the base-class constructor. TAO organizes endpoints as linked lists and the next_ data member points to the next endpoint in the chain of endpoints.

**Pluggable-Endpoint is_equivalent() Function**

**Synopsis**

```c++
CORBA::Boolean is_equivalent (const TAO_Endpoint* other_endpoint)
```

This function compares two endpoints and determines whether they are equivalent. The semantics of what equivalence means is left up to each protocol. The UIOP endpoint simply compares the rendezvous points of the two endpoints (after ensuring they are both UIOP endpoints) as shown below:

```c++
CORBA::Boolean
TAO_UIOP_Endpoint::is_equivalent (const TAO_Endpoint* other_endpoint)
{
    TAO_Endpoint* endpt = ACE_const_cast (TAO_Endpoint*,
                                         other_endpoint);

    TAO_UIOP_Endpoint* endpoint = dynamic_cast<TAO_UIOP_Endpoint*> (endpt);

    if (endpoint == 0)
        return 0;
    return 0;
```
return ACE_OS::strcmp (this->rendezvous_point (),
    endpoint->rendezvous_point ()) == 0;
}

The `rendezvous_point()` member function is a UIOP endpoint helper function that simply returns the rendezvous point (a string) from the ACE address object.

### Pluggable-Endpoint next() Function

**Synopsis**

```cpp
TAO_Endpoint* next (void)
```

This function is used to iterate through the linked lists of endpoints in a profile. The existing implementations all simply return the `next_data` member.

### Pluggable-Endpoint addr_to_string() Function

**Synopsis**

```cpp
int addr_to_string (char* buffer, size_t length)
```

This function is used to convert the endpoint’s address to a string. If the supplied buffer is too small, as indicated by the `length` argument, `addr_to_string()` should return -1. A return of 0 indicates successful operation. This function can be called by multiple threads, therefore it should be reentrant. The UIOP endpoint simply copies the rendezvous point into the buffer after first checking if there is enough space as shown below:

```cpp
int TAO_UIOP_Endpoint::addr_to_string (char* buffer, size_t length) {
    if (length < (ACE_OS::strlen (this->rendezvous_point ()) + 1))
        return -1;

    ACE_OS::strcpy (buffer, this->rendezvous_point ());

    return 0;
}
```

### Pluggable-Endpoint duplicate() Function

**Synopsis**

```cpp
TAO_Endpoint* duplicate (void)
```
Developing Pluggable Protocols

This function dynamically allocates a new endpoint that is an exact copy of the existing endpoint. The UIOP endpoint implements this function by passing the same ACE address to the constructor of the new object as shown below:

```c
TAO_Endpoint *
TAO_UIOP_Endpoint::duplicate (void)
{
    TAO_UIOP_Endpoint* endpoint = 0;
    ACE_NEW_RETURN (endpoint,
        TAO_UIOP_Endpoint (this->object_addr_,
            this->priority ()),
        0);

    return endpoint;
}
```

### Pluggable-Endpoint hash() Function

**Synopsis**

CORBA::ULong hash (void)

This function returns a hash for this endpoint. TAO uses this hash function to efficiently organize and locate endpoints. The UIOP endpoint uses the ACE::hash_pjw() function to generate a hash from the rendezvous point:

```c
CORBA::ULong
TAO_UIOP_Endpoint::hash (void)
{
    if (this->hash_val_ != 0)
        return this->hash_val_;

    ACE_GUARD_RETURN (TAO_SYNCH_MUTEX,
        guard,
        this->addr_lookup_lock_,
        this->hash_val_);

    // .. DCL
    if (this->hash_val_ != 0)
        return this->hash_val_;

    this->hash_val_ =
        ACE::hash_pjw (this->rendezvous_point ());
}
```

return this->hash_val_;
16.4.5 The Profile

The TAO_Profile is the implementation of the protocol details in an IOR. This information is created by servers when publishing an object reference. A profile object is created for each acceptor active in a server. On the client side, the list of profiles is extracted from an IOR and compared against the list of available connectors. As long as the client has a connector supporting one of the protocols specified in the profile, a connection can be established.

The interface for the abstract base-class TAO_Profile is shown below:

```cpp
class TAO_Export TAO_Profile
{
    public:
        TAO_Profile (CORBA::ULong tag,
                     TAO_ORB_Core* orb_core,
                     const TAO_GIOP_Message_Version& version);
        virtual ~TAO_Profile (void);
        CORBA::ULong tag (void) const;
        const TAO_GIOP_Message_Version &version (void) const;
        TAO_ORB_Core *orb_core (void) const;
        CORBA::ULong _incr_refcnt (void);
        CORBA::ULong _decr_refcnt (void);
        void forward_to (TAO_MProfile* mprofiles);
        TAO_MProfile* forward_to (void);
        const TAO_Tagged_Components& tagged_components (void) const;
        TAO_Tagged_Components& tagged_components (void);
        void add_tagged_component (const IOP::TaggedComponent& component
                                   ACE_ENV_ARG_DECL);
        CORBA::Short addressing_mode (void) const;
        const TAO::ObjectKey &object_key (void) const;
        TAO::ObjectKey *_key (void) const;
        CORBA::Boolean is_equivalent (const TAO_Profile* other_profile);
        IOP::TaggedProfile *create_tagged_profile (void);
    virtual char object_key_delimiter (void) const = 0;
};
```

Definitions for these functions are supplied for you by the TAO_Profile class. They are used to support identification and reference counting for memory management. The `forward_to()` accessor and modifier are used in conjunction with the stub object on the client side to retain forwarding information.

The `tagged_components()` accessor functions provide the means to inspect or modify the vendor-specific components of a profile, such as the ORB type, code sets, and priority.

```cpp
virtual char object_key_delimiter (void) const = 0;
```
These pure virtual functions define the behavior essential to profile objects. Each of these functions is discussed in detail in the following pages.
The remainder of the `TAO_Profile` class definition consists of the attributes and internal functions.
Developing Pluggable Protocols

Figure 16-4 shows the UML class diagram illustrating the relationships between the profile, endpoint and acceptor classes.

![UML diagram](image)

For a full discussion as to the contents of the profile class, please refer to the CORBA GIOP and IIOP specs or *IIOP Complete*.

16.4.5.1 Member Functions in Detail

The functions defined in the `TAO_Profile` class are described below.

**Pluggable-Profile Constructor**

A `TAO_Profile` object uses the same protocol tag value as all the related protocol objects. Unlike other protocol support classes, `TAO_Profile` objects are created as needed throughout the life of a client or server. Therefore, your pluggable profile may provide several constructors to serve different needs. The protocol-specific profile classes supplied with TAO actually provide separate constructors, as shown below for IIOP.
The first constructor takes an ACE_Addr-derived address specification and extracts from that any protocol-specific details. In the case of the IIOP profile, this is the host name and port number:

```
TAO_IIOP_Profile (const ACE_INET_Addr& addr,
                 const TAO::ObjectKey& object_key,
                 const TAO_GIOP_Message_Version& version,
                 TAO_ORB_Core* orb_core);
```

The second IIOP constructor is useful when the address elements are pre-marshalled. This form is slightly more efficient than the first form because it does not need to extract the data from the address object:

```
TAO_IIOP_Profile (const char* host,
                 CORBA::UShort port,
                 const TAO::ObjectKey& object_key,
                 const ACE_INET_Addr& addr,
                 const TAO_GIOP_Message_Version& version,
                 TAO_ORB_Core* orb_core);
```

The third IIOP constructor accepts only a pointer to the ORB core and initializes the remaining attributes to their default value:

```
TAO_IIOP_Profile (TAO_ORB_Core* orb_core);
```

### Pluggable-Profile decode_profile() Function

**Synopsis**

```
int decode_profile (TAO_InputCDR& cdr)
```

The `decode_profile()` function is another mechanism for initializing the profile object, this time from an input CDR stream. This initializer is used in the process of extracting a CORBA::Object from a CDR stream. Through that process, the connector registry is called upon to determine the correct profile, if available, then to create a new profile object and initialize it through the TAO_Profile::decode() function which delegates the call to a protocol specific decode_profile().

Though this function is responsible for validating the version number, endpoint address, and object key, it is sufficient to communicate failure by returning a –1 from the function rather than throwing an exception.
Developing Pluggable Protocols

**Pluggable-Profile encode_profile() Function**

**Synopsis**

`int encode_profile (TAO_OutputCDR& stream) const`

The `encode()` function performs the inverse operation of `decode_profile()`. It writes the contents of the profile object to the provided stream.

**Pluggable-Profile hash() Function**

**Synopsis**

`CORBA::ULong hash (CORBA::ULong max)`

The `hash()` function is used to compute a hash-table index value, based on the contents of the profile. This represents the implementation behind the `CORBA::Object::_hash()` operation. The provided `max` value represents the upper bound for hash values.

You may use any algorithm you wish to compute the hash value. For instance, `ACE::hash_pjw()` works well to hash strings.

**Pluggable-Profile object_key_delimiter() Function**

**Synopsis**

`char object_key_delimiter (void) const`

The object-key delimiter is the symbol used by a protocol to separate the address portion of an IOR from the object key. For the IIOP and SHMIOP protocols, the object-key delimiter is a forward slash `/`, but since this symbol can be part of an address specification in UIOP, the UIOP delimiter is a vertical bar `|`. You must implement this correctly according to the constraints of your custom protocol.

For efficiency, the existing profile classes declare the actual object-key delimiter value as a “public static const char.”

This virtual function is required to support the polymorphic behavior in the `TAO_Profile` class.

**Pluggable-Profile parse_string_i() Function**

**Synopsis**

`void parse_string_i (const char* string ACE_ENV_ARG_DECL)`

This function is used to initialize a profile object after it is created by the connector. The `string` argument is of the form used by the `-ORBListenEndpoints` and `-ORBInitRef` options. A well-formed string contains a least an endpoint specification and an object key. These may
optionally be preceded by a version number. For example, “1.2@myhost:1234/obj_key” might be passed to the TAO_IOP_Profile constructor, or “/tmp/rendezvous|obj_key” might be used to initialize a TAO/UIOP_Profile. The protocol prefix is not supplied, since it was used to identify the correct protocol in the first place.

For instance, when TAO_* Connector::make_mprofile() is called and the correct type of profile object is created, the endpoint description is supplied to parse_string_i(). If the provided string is malformed, and an exception using the ACE_THROW macro.

The following code is from the TAO/UIOP_Profile:

```c
void
TAO/UIOP_Profile::parse_string (const char* string
ACE_ENV_ARG_DECL)
{
  if (!string || !*string) {
    ACE_THROW (CORBA::INV_OBJREF {
      CORBA::SystemException::tao_minor_code (CORBA::COMPLETED_NO));
  }

  if (isdigit (string [0]) &&
    string[1] == '.' &&
    isdigit (string [2]) &&
    string[3] == '@') {
    // @@ This may fail for non-ascii character sets [but take that
    // with a grain of salt]
    this->version_.set_version ((char) (string [0] - '0'),
                            (char) (string [2] - '0'));
    string += 4;
    // Skip over the "N.n@"
  }

  if (this->version_.major != TAO_DEF_GIOP_MAJOR ||
      this->version_.minor  > TAO_DEF_GIOP_MINOR) {
    ACE_THROW (CORBA::INV_OBJREF (}
Developing Pluggable Protocols

CORBA_SystemException::tao_minor_code (  
    CORBA::COMPLETED_NO));
}

The above code uses assumptions to optimize its determination of the version. Since UIOP has only a single major version, and less than 10 minor versions, the initial sequence of characters must be exactly “M.n@”, with the M being exactly the major version and n being less than or equal to the highest minor version number. It is an error to have any other number for M or n, and no other string is recognized as a version number.

A stronger test may be to look for the version separator, ‘@’, and disqualify anything to the left that does not resolve to an allowed version. As we see below, a malformed version number is prepended to the rendezvous string:

    // Pull off the "rendezvous point" part of the objref
    // Copy the string because we are going to modify it...
    CORBA::String_var copy (string);

    char* start = copy.inout ();
    char* cp = ACE_OS::strchr (start, this->object_key_delimiter_);

    if (cp == 0) {
        ACE_THROW (CORBA::INV_OBJREF (  
            CORBA_SystemException::tao_minor_code (  
                CORBA::COMPLETED_NO));
    // No rendezvous point specified
    }

    It is an error to not include both an address and an object key in the string. This is determined by the existence of the object-key delimiter. The final step is to initialize the profile’s object-address and object-key attributes with the two halves of the remaining string.

    CORBA::ULong length = cp - start;

    CORBA::String_var rendezvous = CORBA::string_alloc (length);

    ACE_OS::strncpy (rendezvous.inout (), start, length);
    rendezvous[length] = '\0';

    if (this->endpoint_.object_addr_.set (rendezvous.in ()) != 0) {
16.4 Details of the Pluggable-Protocol Framework

ACE_THROW (CORBA::INV_OBJREF (  
   CORBA_SystemException::tao_minor_code (  
      TAO_DEFAULT_MINOR_CODE,  
      EINVAL),  
   CORBA::COMPLETED_NO));

start = ++cp; // increment past the object-key separator

TAO::ObjectKey ok;  
TAO::ObjectKey::decode_string_to_sequence (ok, start);

this->orb_core ()->object_key_table ().bind (ok,  
   this->ref_object_key_);

The final test is to determine that the address portion of the string is not of zero length. At this point, there is no test that the string is indeed a valid address. That happens later, when the address is used. Likewise, no validity test is performed on the object-key string. Returning without throwing an exception indicates success.

Pluggable-Profile to_string() Function

Synopsis
char* to_string (ACE_ENV_SINGLE_ARG_DECL)

TAO_Profile::to_string() generates an IOR string representing the profile object. This is essentially the inverse of the parse_string() operation, but with the addition of the “loc” prefix at the beginning of the string. For instance, a UIOP IOR for version 1.1, at the rendezvous point /tmp/uiopmeet and object-key myobj would yield the string:

uioploc://1.1@/tmp/uiopmeet|myobj

The UIOP implementation always generates a string representing the profile. You may, however wish to detect error conditions and communicate them using the ACE_THROW macro.

Pluggable-Profile endpoint() Function

Synopsis
TAO_Endpoint* endpoint (void)

This function returns the head endpoint in the endpoint chain contained within this profile. The existing protocols store a protocol-specific endpoint pointer in the derived profile class.
Developing Pluggable Protocols

Pluggable-Profile endpoint_count() Function

Synopsis

```c
size_t endpoint_count (void)
```

This function returns the number of endpoints contained within this profile.

Pluggable-Profile encode_endpoints() Function

Synopsis

```c
int encode_endpoints (void)
```

This function encodes the profile’s endpoints into the tagged components, storing the result in the tagged_components_data member of the TAO_Profile base class.

16.4.6 The Protocol Factory

The protocol factory is responsible for creating the acceptors used by servers and the connectors used by clients. The protocol factory also provides the symbol used to separate options from addresses when parsing endpoint specifications, and the prefix that is the key for determining if a given protocol factory is appropriate for providing the acceptor and connector for a specified protocol.

The protocol factories in TAO are owned by the resource factory. The protocol factories themselves are lightweight objects, serving only to create the object that performs the real work of managing endpoints for the ORB. Here is the public interface of the TAO_Protocol_Factory base class:

```c
class TAO_Export TAO_Protocol_Factory : public ACE_Service_Object
{
    public:
        TAO_Protocol_Factory (CORBA::ULong tag);
        virtual ~TAO_Protocol_Factory (void);
        virtual int init (int argc, char* argv[]);
        CORBA::ULong tag (void) const;
        virtual int match_prefix (const ACE_CString& prefix);
        virtual const char* prefix (void) const;
        virtual char options_delimiter (void) const;
        virtual TAO_Acceptor* make_acceptor (void);
        virtual TAO_Connector* make_connector (void);
        virtual int requires_explicit_endpoint (void) const = 0;

    private:
        CORBA::ULong tag_;
};
```
Unlike the other classes presented in this chapter, most of the functions of the protocol factory have default implementations. For the most part, the default implementations return failure indicators.

Pluggable-protocol factories must derive from `TAO_Protocol_Factory`, which is a service object managed by the service configurator. The base class is defined in `<tao/Protocol_Factory.h>`. Because protocol factories are service objects, it is necessary to include the stand-alone static factory functions for creation of the factory object itself. See Chapter 18 for an in-depth discussion of the service configurator and how to integrate with it. Later in this chapter, we cover the factory functions used by the service-configurator framework with regard to the pluggable-protocol framework.

We now detail how a protocol-specific factory is compiled and/or linked and instantiated in your application, using the `TAO_UIOP_Protocol_Factory` as an example. The following macros appear in the header file `<tao/Strategies/UIOP_Factory.h>`:

```c
ACE_STATIC_SVC_DECLARE (TAO_UIOP_Protocol_Factory)
ACE_FACTORY_DECLARE (TAO_Strategies, TAO_UIOP_Protocol_Factory)
```

The first macro declares a data structure that is required for registering a statically-linked service into the service configurator. This macro should be used in the header file where the service is declared. Its only argument is the name of the class that implements the service.

The second macro declares the factory function used to create dynamically-loadable services. Once the service implementation is dynamically loaded, the service configurator uses a factory function to create the object. This macro declares such a factory function with the proper export specification.

The preprocessor expands these macros to the following C++ declarations respectively:

```c
extern ACE_Static_Svc_Descriptor ace_svc_desc_TAO_UIOP_Protocol_Factory;
extern "C" TAO_Strategies_Export ACE_Service_Object * _make_TAO_UIOP_Protocol_Factory (ACE_Service_Object_Exterminator *);
```
The following macros appear in `<tao/Strategies/UIOP_Factory.cpp>`, which correspond to the macros in the header file:

ACE_STATIC_SVC_DEFINE (TAO_UIOP_Protocol_Factory,
    ACE_TEXT ("UIOP_Factory"),
    ACE_SVC_OBJ_T,
    &ACE_SVC_NAME (TAO_UIOP_Protocol_Factory),
    ACE_Service_Type::DELETE_THIS |
    ACE_Service_Type::DELETE_OBJ,
    0)

ACE_FACTORY_DEFINE (TAO_Strategies, TAO_UIOP_Protocol_Factory)

The first macro creates an ACE_Static_Svc_Descriptor, which is a structure used by the service-configurator framework. The second macro defines both the factory function and the function used to clean up the service object.

These two macros expand to the following C++ code, respectively:

ACE_Static_Svc_Descriptor ace_svc_desc_TAO_UIOP_Protocol_Factory =
{    "UIOP_Factory",
265,
    &make_TAO_UIOP_Protocol_Factory,
0x3,
0 }

and

extern "C"
void _gobble_TAO_UIOP_Protocol_Factory(void* p)
{
    ACE_Service_Object* _p = ACE_reinterpret_case(ACE_Service_Object, p);
    ACE_ASSERT(_p != 0);
    delete p;
}

extern "C"
ACE_Service_Object* _make_TAO_UIOP_Protocol_Factory (ACE_Service_Object_Exterminator* gobbler)
{
    ACE_TRACE("TAO_UIOP_Protocol_Factory");
    if (gobbler != 0)
        *gobbler =
        (ACE_Service_Object_Exterminator) _gobble_TAO_UIOP_Protocol_Factory;
return new TAO_UIOP_Protocol_Factory;
}

You can see within this last function the exact location where the protocol-specific factory object is instantiated. A pointer to this new factory object is returned to the service-configurator framework, where it is subsequently stored in the ACE_Service_Repository, and later in a collection called the protocol-factory set. One additional side effect of this function is to set the output parameter gobbler to the address of the first function that is used by the service configurator to destroy the factory object.

There is quite a bit that goes on behind the scenes to get your protocol-specific factory object up and running, so that it can then be used by the connector and acceptor registries to create their respective protocol-specific components.

Figure 16-5 Pluggable-Protocol Framework and Service Configurator
The following two sections provide a detailed explanation as to how this all comes about. While fully understanding the process of protocol factory initialization is not required to implement a pluggable protocol, it does give you the complete picture of this stage where the TAO internals interact with your protocol-specific classes. Figure 16-5 is a UML class diagram showing the relationships between the service-configurator classes and those of the pluggable-protocol framework.

16.4.6.1 Static Loading

For static loading of protocols, the basic sequence of events for instantiating and “loading” the protocol factory is discussed below.

If the following include statement is placed before `main()`, all protocol factories in the strategies library are instantiated before `main()` is called:

```cpp
#include "tao/Strategies/advanced_resource.h"
```

This is due to some static global objects in the strategies library that are instantiated as a result of including the `advanced_resource.h` file before `main()`. The static factory functions for each protocol, as shown above for UIOP, are used by the service-configurator framework to instantiate the protocol-specific protocol-factory objects. These objects are not only instantiated, but are also put into the service repository for later reference.

Once `main()` executes and `CORBA::ORB_init()` is called, the initialization of the protocol factories begins. First, the `TAO_IIO_Protocol_Factory` is instantiated through its static factory function, and is added to the service repository. The service configurator then processes the service-configurator file to determine which protocols to set up for loading. “Load” in this sense means to take from the service repository and put into another collection called the protocol-factory set.

Based on what the service configurator sees in the contents of the service-configurator file, it sets up parts of the contents of the protocol-factory set with the names of the particular protocol that is officially loaded by the resource factory later in the ORB initialization. Not until the advanced resource factory is instructed to initialize the protocol factories, is the protocol-factory set filled in with the particular protocol-specific factories. After this point, those protocols are considered “loaded” (i.e., present in the
protocol-factory set and ready to be used to instantiate protocol-specific acceptors and connectors).

To statically link and load your custom protocol, you should mimic what is done by the SSLIOP. The following macros are in SSLIOP_Factory.h:

\[
\begin{align*}
\text{ACE_STATIC_SVC_DECLARE_EXPORT} &\ (\text{TAO_SSLIOP,TAO_SSLIOP_Protocol_Factory}) \\
\text{ACE_STATIC_SVC_REQUIRE} &\ (\text{TAO_SSLIOP_Protocol_Factory})
\end{align*}
\]

These macros expand to:

\[
\begin{align*}
\text{extern TAO_SSLIOP Export ACE_Static_Svc_Descriptor} \\
\text{ace_svc_desc_TAO_SSLIOP_Protocol_Factory;}
\end{align*}
\]

\[
\begin{align*}
\text{class ACE_Static_Svc_TAO_SSLIOP_Protocol_Factory} \\
\{ \\
\text{public:} \\
\text{ACE_Static_Svc_TAO_SSLIOP_Protocol_Factory()} \\
\{ \\
\text{ACE_Service_Config::static_svc() \rightarrow insert (} \\
\&\text{ace_svc_desc_TAO_SSLIOP_Protocol_Factory);} \\
\} \\
\}; \\
\text{static ACE_Static_Svc_TAO_SSLIOP_Protocol_Factory} \\
\text{ace_static_svc_TAO_SSLIOP_Protocol_Factory;}
\end{align*}
\]

For static linking, SSLIOP_Factory.h is included before the main() function, causing the ACE_Static_Svc_Descriptor to be loaded into the service configurator’s collection of descriptors for later processing during initialization. For this to work, the SSLIOP_Factory.h file must have the following line at the top:

\[
\text{#include } \text{"ace/Service_Config.h"}.
\]

To statically load your custom protocol, mimic what the SSLIOP protocol-factory code does.

### 16.4.6.2 Dynamic Loading

In the case of dynamic loading of protocols, the actions taken are basically the same as in the case of static loading, but they occur in a different order.

No protocol factories are instantiated before main(), because we are not including the advanced_resource.h header file.
As part of ORB initialization, the IIOP protocol factory is the first to be instantiated and added to the service repository. It is during the processing of the service-configurator file that the dynamic directives are handled, causing the TAO_Strategies library (or whatever library is specified in the directive) to be loaded, then the protocol-specific factories to be instantiated via the factory functions defined through the use of the macros.

Finally, the protocol factory is loaded into the service repository. As in the static case, when the resource factory is mandated to initialize the protocol factories, the appropriate protocols are loaded into the protocol-factory set for later use by the acceptor and connector registry.

The initialization of the resource factories puts half-filled entries into the protocol-factory set. At this point, these entries contain only the key string that is the name of the protocol factory. Later, when the resource factory is instructed to call init_protocol_factories(), the rest of the parts of the entries in the protocol-factory set (in this case the pointer to the concrete factory itself) are filled in. The resource factory gets access to the concrete protocol factory from the service repository via the ACE_Dynamic_Service class.

16.4.6.3 The Bigger Picture

Viewing this whole initialization process from a higher level, there are really five main containers that are pertinent to the initialization of the protocol-factory-related classes. These are:

1. Service repository
2. Protocol-factory set
3. Collection of ACE_Static_Svc_Descriptors
4. Connector registry
5. Acceptor registry

The ACE_Service_Config class and the resource factories work together to populate and process these various containers to create a protocol-factory set. That can be used by other parts of the ORB internals to begin making the protocol-specific components that do the business work of the establishing connections.
Figure 16-6 shows a basic UML class diagram illustrating the relationships between the protocol-factory class and other classes in the ORB internals.

![Figure 16-6 Protocol-Factory Relationships](image_url)

### 16.4.6.4 The Member Functions in Detail

**Pluggable-Protocol-Factory Constructor**

Since the protocol factory is derived from `ACE_Service_Object`, the constructor and destructor defer most of the initialization and cleanup responsibilities to the `init()` and `fini()` functions. This constructor is only used to initialize the `tag_` data member as shown below:

```cpp
TAO_Protocol_Factory::TAO_Protocol_Factory (CORBA::ULong tag)
  : tag_ (tag)
{
}
```
The following TAO_UIOP_Protocol constructor illustrates how to pass your protocol-specific tag value to the base constructor:

```cpp
TAO_UIOP_Protocol_Factory::TAO_UIOP_Protocol_Factory (void)
    : TAO_Protocol_Factory (TAO_TAG_UIOP_PROFILE),
      major_ (TAO_DEF_GIOP_MAJOR),
      minor_ (TAO_DEF_GIOP_MINOR)
{
}
```

### Pluggable-Protocol-Factory init() Function

**Synopsis**

```cpp
int init (int argc, char* argv[])```

This `init()` function is called by the service configurator when the protocol-factory object is initialized. The options provided in the service directive are passed to `init()` as `argc` and `argv`. The interpretation of the arguments is entirely up to you. Currently, SHMIOP and SSLIOP are the only two supplied protocols that process arguments. The remaining protocols (IIOP, DIOP, and UIOP) simply supply a 0 return value. The base-class `init()` function returns –1.

### Pluggable-Protocol-Factory make_acceptor() Function

**Synopsis**

```cpp
TAO_Acceptor* make_acceptor (void)```

As new acceptors are needed by the ORB, `make_acceptor()` is called. It is the responsibility of the protocol factory to construct a new acceptor instance and return a pointer to it if the allocation worked properly, or to return `NULL` if the allocation failed.

The protocol factories supplied with TAO make all of their acceptors by allocating them from the heap using the `ACE_NEW_RETURN` macro. It is possible to supply arguments to the acceptor’s constructor. However, since the signature of the `-ORBListenEndpoints` option allows for options to be provided to the acceptor’s `open()` function, it is more appropriate to use a parameterless constructor that performs only the basic initialization of the acceptor, relying on the options provided to the acceptor, when it is opened, to tune its behavior. A UIOP example of this function is shown below:

```cpp
TAO_Acceptor*
TAO_UIOP_Protocol_Factory::make_acceptor (void)
{
    TAO_Acceptor* acceptor = 0;
```
ACE_NEW_RETURN (acceptor,
    TAO_UIOP_Acceptor,
    0);

    return acceptor;
}

If `make_acceptor()` is not overridden, the base protocol-factory object returns a null pointer.

**Pluggable-Protocol-Factory make_connector() Function**

**Synopsis**

TAO_Connector* make_connector (void)

Similar to the `make_acceptor()` function, the `make_connector()` function is responsible for instantiating new protocol-specific connectors. The protocol factory should simply allocate a new connector, then return a pointer to it, without attempting any special initialization. The connector’s `open()` function is called later to perform initialization tasks.

In a manner similar to making acceptors, your protocol factory should return a null pointer if it is unable to create a new connector instance. A UIOP example of this function is shown below:

```cpp
TAO_Connector *
TAO_UIOP_Protocol_Factory::make_connector (void)
{
    TAO_Connector* connector = 0;

    ACE_NEW_RETURN (connector,
        TAO_UIOP_Connector,
        0);

    return connector;
}
```

The base class always returns a null pointer from `make_connector()`.

**Pluggable-Protocol-Factory match_prefix() Function**

**Synopsis**

int match_prefix (const ACE_CString& prefix)

This function is used by TAO to determine which factory serves the protocol denoted by the prefix string. Your protocol factory must return 1 if the prefix
Developing Pluggable Protocols

in question matches the prefix defined for your protocol. The comparison must be case-insensitive.

This prefix comes from the prefix to a -ORBListenEndpoints or -ORBInitRef definition. For instance, “iiop” is the prefix obtained from an endpoint such as “iiop://1.1@myhost:1234”.

Here is an example of the match_prefix() function, from the UIOP protocol factory:

```c
int
TAO_UIOP_Protocol_Factory::match_prefix (const ACE_CString &prefix)
{
    // Check for the proper prefix for this protocol.
    return (ACE_OS::strcasecmp (prefix.c_str (), ::prefix_) == 0);
}
```

The base protocol factory always fails to match.

**Pluggable-Protocol-Factory options_delimiter() Function**

**Synopsis**

```c
char options_delimiter (void) const
```

The options delimiter is the character that denotes the end of an endpoint address and the start of the initialization options. In the case of IIOP, the options delimiter is ‘/’. This symbol is not appropriate for UIOP, since a slash is used as part of the address, so UIOP uses ‘|’ as the options delimiter. A UIOP example of this function is shown below:

```c
char
TAO_UIOP_Protocol_Factory::options_delimiter (void) const
{
    return '|';
}
```

The base class returns a null character if no options delimiter is specified by your protocol factory.

**Pluggable-Protocol-Factory prefix() Function**

**Synopsis**

```c
const char* prefix (void) const
```

This function supplies the string representing the protocol supported by this factory. The base protocol factory returns a null string for prefix().
The prefix must be unique for each protocol, but this uniqueness is not tested. The prefix strings used by the TAO-supplied protocols are iiop, uiop, diop, and shmiop. A UIOP example of this function is shown below:

```c
const char*
TAO_UIOP_Protocol_Factory::prefix (void) const
{
    return ::prefix_;    
}
```

### Pluggable-Protocol-Factory requires_explicit_endpoint() Function

**Synopsis**

int requires_explicit_endpoint (void) const

When a server is started without any -ORBListenEndpoints options, TAO iterates over the protocol-factory repository, requesting an acceptor from each factory for which this function returns 0. These acceptors are then opened with the open_default() function.

Some protocols, such as UIOP, involve file system artifacts that are not cleaned up if the server terminates abruptly. To avoid polluting the file system with stale rendezvous points, the UIOP protocol factory returns 1 from the requires_explicit_endpoint(), requiring the server to be started with an explicit -ORBListenEndpoints option that includes at least “uiop://” in the endpoint list. A return value of 1 from this function does not mean that open_default() is not called for UIOP acceptors, only that UIOP must be explicitly requested as an endpoint.

This function is pure virtual in the base class.

### 16.4.7 The Transport

TAO separates the role of event notification related to a communication mechanism from the role of message transmission on that mechanism. In 16.4.3, we cover the connection handler, which deals with connection management and event notification. Here we look at the message-transmission half of the relationship.

The activities relating to the sending and receiving of messages are the responsibility of classes derived from TAO_Transport. Objects of these types are constructed by connection handlers on both the client side and the server side.
The transport classes serve as the interface for incoming messages to be handled by GIOP, and also for outgoing messages resulting from invocation requests.

Figure 16-7 shows the relationship between the protocol-specific transport class and its base class. The interface for the abstract base-class TAO_Transport is shown below:

```cpp
class TAO_Export TAO_Transport
{
public:

  TAO_Transport (CORBA::ULong tag,
                  TAO_ORB_Core *orb_core);
  virtual ~TAO_Transport (void);

  CORBA::ULong tag (void) const;
};
```
16.4 Details of the Pluggable-Protocol Framework

TAO_ORB_Core *orb_core (void) const;
TAO_Transport_Mux_Strategy *tms (void) const;
TAO_Wait_Strategy *wait_strategy (void) const;
int handle_output (void);

int bidirectional_flag (void) const;
void bidirectional_flag (int flag);

void cache_map_entry (TAO_Transport_Cache_Manager::HASH_MAP_ENTRY *entry);
TAO_Transport_Cache_Manager::HASH_MAP_ENTRY *cache_map_entry (void);

size_t id (void) const;
void id (size_t id);

TAO::Connection_Role opened_as (void) const;
void opened_as (TAO::Connection_Role);

unsigned long purging_order (void) const;
void purging_order (unsigned long value);

int queue_is_empty (void);
void provide_handler (TAO_Connection_Handler_Set &handlers);
virtual int register_handler (void);

virtual ssize_t send (iovec *iov, int iovcnt,
size_t &bytes_transferred,
const ACE_Time_Value *timeout = 0) = 0;
virtual ssize_t recv (char *buffer,
size_t len,
const ACE_Time_Value *timeout = 0) = 0;
bool idle_after_send (void);
bool idle_after_reply (void);
virtual void close_connection (void);

virtual int messaging_init (CORBA::Octet major,
CORBA::Octet minor) = 0;
virtual int tear_listen_point_list (TAO_InputCDR &cdr);
ACE_Event_Handler::Reference_Count add_reference (void);
ACE_Event_Handler::Reference_Count remove_reference (void);

virtual TAO_Pluggable_Messaging * messaging_object (void) = 0;

virtual ACE_Event_Handler * event_handler_i (void) = 0;
bool is_connected (void) const;
bool post_open (size_t id);
TAO_Connection_Handler * connection_handler (void);
TAO_OutputCDR &out_stream (void);

protected:
virtual TAO_Connection_Handler * connection_handler_i (void) = 0;

public:

int generate_locate_request (TAO_Target_Specification &spec,
   TAO_Operation_Details &opdetails,
   TAO_OutputCDR &output);

virtual int generate_request_header (TAO_Operation_Details &opd,
   TAO_Target_Specification &spec,
   TAO_OutputCDR &msg);

int recache_transport (TAO_Transport_Descriptor_Interface* desc);

virtual int handle_input (TAO_Resume_Handle &rh,
   ACE_Time_Value *max_wait_time = 0,
   int block = 0);

enum
{
   TAO_ONEWAY_REQUEST = 0,
   TAO_TWOWAY_REQUEST = 1,
   TAO_REPLY
};

virtual int send_request (TAO_Stub *stub,
   TAO_ORB_Core *orb_core,
   TAO_OutputCDR &stream,
   int message_semantics,
   ACE_Time_Value *max_time_wait) = 0;

virtual int send_message (TAO_OutputCDR &stream,
   TAO_Stub *stub = 0,
   int message_semantics = TAO_Transport::TAO_TWOWAY_REQUEST,
   ACE_Time_Value *max_time_wait = 0) = 0;

virtual int send_message_shared (TAO_Stub *stub,
   int message_semantics,
   const ACE_Message_Block *message_block,
   ACE_Time_Value *max_wait_time);

protected:

int parse_consolidate_messages (ACE_Message_Block &bl,
   TAO_Resume_Handle &rh,
   ACE_Time_Value *time = 0);

int parseIncoming_messages (ACE_Message_Block &message_block);

size_t missing_data (ACE_Message_Block &message_block);

virtual int consolidate_message (ACE_Message_Block &incoming,
   ssize_t missing_data,
   TAO_Resume_Handle &rh,
   ACE_Time_Value *max_wait_time);

int consolidate_fragments (TAO_Queued_Data *qd,
   TAO_Resume_Handle &rh);

int consolidate_message_queue (ACE_Message_Block &incoming,
   ssize_t missing_data,
   TAO_Resume_Handle &rh,
16.4 Details of the Pluggable-Protocol Framework

ACE_Time_Value *max_wait_time);
int consolidate_extra_messages (ACE_Message_Block &incoming,
   TAO_Resume_Handle &rh);
int process_parsed_messages (TAO_Queue_Data *qd,
   TAO_Resume_Handle &rh);
TAO_Queue_Data *make_queued_data (ACE_Message_Block &incoming);
int send_message_shared_i (TAO_Stub *stub,
   int message_semantics,
   const ACE_Message_Block *message_block,
   ACE_Time_Value *max_wait_time);
int queue_message_i (const ACE_Message_Block *message_block);

public:
int format_queue_message (TAO_OutputCDR &stream);
int send_message_block_chain (const ACE_Message_Block *message_block,
   size_t &bytes_transferred,
   ACE_Time_Value *max_wait_time = 0);
int send_message_block_chain_i (const ACE_Message_Block *message_block,
   size_t &bytes_transferred,
   ACE_Time_Value *max_wait_time);

void purge_entry (void);
int make_idle (void);
int update_transport (void);
int handle_timeout (const ACE_Time_Value &current_time,
   const void* act);
size_t recv_buffer_size (void);
size_t sent_byte_count (void);

TAO_Codeset_Translator_Base *char_translator (void) const;
TAO_Codeset_Translator_Base *wchar_translator (void) const;

void char_translator (TAO_Codeset_Translator_Base *);
void wchar_translator (TAO_Codeset_Translator_Base *);
void assign_translators (TAO_InputCDR *, TAO_OutputCDR *);
void clear_translators (TAO_InputCDR *, TAO_OutputCDR *);
CORBA::Boolean is_tcs_set() const;
void first_request_sent();
void send_connection_closed_notifications (void);

private:
void send_connection_closed_notifications (void);
int drain_queue (void);
int drain_queue_i (void);

friend class TAO_Block_Flushing_Strategy;

int queue_is_empty_i (void);
int drain_queue_helper (int &iovcnt, iovec iov[]);

friend class TAO_Reactive_Flushing_Strategy;
friend class TAO_Leader_Follower_Flushing_Strategy;
friend class TAO_Transport_Cache_Manager;

int schedule_output_i (void);
int cancel_output_i (void);
void cleanup_queue (size_t byte_count);
void cleanup_queue_i ();
int check_buffering_constraints_i (TAO_Stub *stub, int &must_flush);
int send_synchronous_message_i (const ACE_Message_Block *message_block,
ACE_Time_Value *max_wait_time);
int send_reply_message_i (const ACE_Message_Block *message_block,
ACE_Time_Value *max_wait_time);
int send_synch_message_helper_i (TAO_Synch_Queued_Message &s,
ACE_Time_Value *max_wait_time);
int flush_timer_pending (void) const;
void reset_flush_timer (void);
void report_invalid_event_handler (const char *caller);
int process_queue_head (TAO_Resume_Handle &rh);
int notify_reactor (void);

void send_connection_closed_notifications_i (void);
void process_fragment (TAO_Queue_Data* fragment_message,
TAO_Queue_Data* queueable_message,
CORBA::Octet major,
CORBA::Octet minor,
TAO_Resume_Handle &rh);
void allocate_partial_message_block (void);

ACE_UNIMPLEMENTED_FUNC (TAO_Transport (const TAO_Transport&))
ACE_UNIMPLEMENTED_FUNC (void operator= (const TAO_Transport&))

protected:
CORBA::ULong tag_;
TAO_ORB_Core *orb_core_;
TAO_Transport_Cache_Manager::HASH_MAP_ENTRY *cache_map_entry_;
TAO_Transport_Mux_Strategy *tms_;
TAO_Wait_Strategy *ws_;
int bidirectional_flag_;
TAO::Connection_Role opening_connection_role_;
TAO_Queue_Message *head_;
TAO_Queue_Message *tail_;
TAO_Incoming_Message_Queue incoming_message_queue_;
ACE_Time_Value current_deadline_;
long flush_timer_id_;
TAO_Transport_Timer transport_timer_;
ACE_Lock *handler_lock_;
size_t id_;
unsigned long purging_order_;
size_t recv_buffer_size_;
size_t sent_byte_count_;

Developing Pluggable Protocols
The TAO_Transport interface contains a large number of virtual functions, many of which have default implementations. Each of these virtual functions is discussed below. There are also many non-virtual member functions for use in the TAO_Transport and derived classes. We discuss some of those functions where they are used in the virtual function implementations below.

The key pattern used in the TAO_Transport class is the Template Method pattern. The template functions of the TAO_Transport class outline the algorithm (steps) for sending the data, but the actual implementation alternates between the base-class and the derived-class functions.

### 16.4.7.1 Member Functions in Detail

#### Pluggable-Transport Constructor

Like the other components in the pluggable-protocol framework, the transport is constructed with the tag for that protocol. The constructor is also passed a pointer to the ORB core that owns the transport.

The base TAO_Transport class implements components related to the wait (see 22.2.3) and multiplexing (see 22.2.2) strategies for the client. It uses the provided ORB core to locate the client strategy factory, from which it obtains the appropriate strategy objects. Refer to Chapter 22 for information on using the default client strategy factory to configure these strategies.

Your constructor should do little more than supply a tag value to the base and initialize the attributes. The transport object is constructed as the result of a server accepting a new connection or a client successfully completing a connection request.

Here is the UIOP transport constructor, which is passed a pointer to the connection handler that created it and a flag that determines whether the GIOP-lite messaging protocol should be used:
Developing Pluggable Protocols

```cpp
TAO_UIOP_Transport::TAO_UIOP_Transport (TAO_UIOP_Connection_Handler* handler,
   TAO_ORB_Core* orb_core,
   CORBA::Boolean flag)
   :
   TAO_Transport (TAO_TAG_UIOP_PROFILE,
   orb_core)
   , connection_handler_ (handler)
   , messaging_object_ (0)
{
   if (flag) {
      // Use the lite version of the protocol
      ACE_NEW (this->messaging_object_,
               TAO_GIOP_Message_Lite (orb_core));
   }
   else {
      // Use the normal GIOP object
      ACE_NEW (this->messaging_object_,
               TAO_GIOP_Message_Base (orb_core));
   }
}
```

The `messaging_object_` data member is used to store a pointer to a `TAO_Pluggable_Messaging` object, which gives the transport the needed visibility to the TAO messaging layer.

### Pluggable-Transport close_connection() Function

**Synopsis**

```cpp
void close_connection (void)
```

Since this behavior is common between both clients and servers, it should be implemented in the base transport class. This function can be called as the result of an error, while sending a request or some other message, or as the result of sending or receiving a close-connection GIOP message.

### Pluggable-Transport send() Function

**Synopsis**

```cpp
virtual ssize_t send_i (iovec* iov,
   size_t& bytes_transferred,
   const ACE_Time_Value* timeout = 0)
```

This function is the key protocol-specific function that gets called from the base-class sending algorithm. Most functionality up to this point, as far as the send is concerned, has been protocol-independent. At the point of this call, the `iovec` list contains a fully-formatted GIOP message that is ready for the protocol-specific transmission.
The `send()` function transmits the contents of the `iovec` list. It is expected to wait until either the entire block is sent or until the time limit is reached. It is part of the interface to the underlying transport layer, so the implementation depends upon the implementation of the underlying streaming class. The `bytes_transferred` reference should be set to the number of bytes successfully transferred before the request blocked.

The simplest way to implement `send()` is by basing the underlying stream on the `ACE_IPC_SAP` family of classes. That way the connection handler, which derives from the `ACE_Svc_Handler` class, provides access to the stream class via the `peer()` function. The peer is of type `ACE_*_Stream`, and should supply a form of `send()` that accepts an `iovec` list and a timeout. We cover the use of these ACE classes in later sections.

Here is an example of such a call from the UIOP transport class:

```c
ssize_t
TAO_UIOP_Transport::send (iovec *iov, int iovcnt,
                          size_t &bytes_transferred,
                          const ACE_Time_Value *max_wait_time)
{
    const ssize_t retval =
        this->connection_handler_->peer ()->sendv (iov,
                                                iovcnt,
                                                max_wait_time);

    if (retval > 0)
        bytes_transferred = retval;

    return retval;
}
```

### Pluggable-Transport send_request() Function

**Synopsis**

```c
send_request (TAOStub* stub,
              TAOORB_Core* orb_core,
              TAOOutputCDR& stream,
              int message_semantics,
              ACE_Time_Value* max_time_wait)
```

This function is called by the upper-level invocation object when a request is to be transmitted to the server. Unless you have an overriding reason to make a customized implementation, your custom protocol can use the exact lines used from one of the currently-implemented protocols. In a future version of TAO, this function implementation will be refactored into the base class as a
normal virtual function, thus providing the capability for concrete derived protocol-specific transport classes to inherit the implementation that is common to all protocols, while at the same time leaving open the option for override. Below is an example of this function, implemented in the TAO/UIOP Protocol class:

```c
int TAO_UIOP_Transport::send_request (TAO_Stub *stub,
    TAO_ORB_Core *orb_core,
    TAO_OutputCDR &stream,
    int message_semantics,
    ACE_Time_Value *max_wait_time)
{
    if (this->ws_->sending_request (orb_core,
        message_semantics) == -1)
        return -1;

    if (this->send_message (stream,
        stub,
        message_semantics,
        max_wait_time) == -1)
        return -1;

    return 0;
}
```

**Pluggable-Transport send_message() Function**

**Synopsis**

```c
virtual int send_message (TAO_OutputCDR& stream,
    TAO_Stub* stub = 0,
    int is_synchronous = 1,
    ACE_Time_Value* max_time_wait = 0)
```

On the client side, when a request is being sent to the server, this function is called by the send_request() transport member function. It serves the sole purpose of using the messaging-level object to begin formatting the outgoing CDR stream. Then this function simply forwards the data along to the sending functions.

On the server side, this function is called by the ServerRequest object after the upcall has reached the servant and the reply is coming back to be sent to the client.

As with the send_request() function, the current TAO protocols all share the exact same implementation. Future releases of TAO will refactor this
common implementation into the base class. When using TAO, your custom protocol can mimic the implementation of the other protocols.

Here is the TAO_UIOP_Transport implementation of send_message():

```c
int
TAO_UIOP_Transport::send_message (TAO_OutputCDR &stream,
    TAO_Stub *stub,
    int message_semantics,
    ACE_Time_Value *max_wait_time)
{
    // Format the message in the stream first
    if (this->messaging_object_->_format_message (stream) != 0)
        return -1;

    // Strictly speaking, should not need to loop here because the
    // socket never gets set to a nonblocking mode ... some Linux
    // versions seem to need it though. Leaving it costs little.

    // This guarantees to send all data (bytes) or return an error.
    ssize_t n = this->send_message_shared (stub,
        message_semantics,
        stream.begin (),
        max_wait_time);

    if (n == -1)
    {
        if (TAO_debug_level)
            ACE_DEBUG ((LM_DEBUG,
                ACE_TEXT("TAO: (%P|%t|%N|%l) closing transport %d after fault
"),
                this->id (),
                ACE_TEXT("send_message ()\n"));

        return -1;
    }

    return 1;
}
```

### Pluggable-Transport send_message_shared() Function

**Synopsis**

```c
int send_message_shared (TAO_Stub* stub,
    int message_semantics,
    const ACE_Message_Block* message_block,
    ACE_Time_Value* max_wait_time)
```

This is the key public-interface non-virtual function. It mandates the sequence of steps needed to send a message (request or reply) out to the peer object.
This function uses the Template Method pattern to get its job done, deferring certain steps to derived classes for protocol-specific behavior, while allowing non-protocol-specific behavior to be executed in the common base-class member functions. This function branches based on the synchronous nature of the request. For the most part you need not be concerned about the internals of this function. You only need to be aware of where the derived class functionality is invoked. We cover these exact points, so that you can provide the protocol-specific function implementations there.

Pluggable-Transport consolidate_message() Function

Synopsis

```c
int consolidate_message (ACE_Message_Block &incoming,
                         ssize_t missing_data,
                         TAO_Resume_Handle &rh,
                         ACE_Time_Value *max_wait_time);
```

This function is invoked when either the message queue tail or the incoming message itself is incomplete. Current protocols just use the base class implementation as the internals of the function are protocol independent.

Pluggable-Transport handle_input() Function

Synopsis

```c
int handle_input (TAO_Resume_Handle &rh,
                  ACE_Time_Value * max_wait_time = 0,
                  int block = 0)
```

This function is the key receipt-related public function that establishes the algorithm (steps) for receiving a message. This function follows the Template Method pattern by deferring some steps to protocol-specific derived classes, while implementing protocol-independent functionality in the base-class member functions. Typically, this function is called directly from the handle_input() function of the connection handler, which was previously dispatched from the reactor. In the case of an activated (active object) connection handler, this function is called from within its thread function svc_i(). As a writer of custom protocols, you need not be overly concerned about the implementation of this function. All you really need to know is that this function triggers a call to your protocol-specific recv() function. All other functionality is protocol-independent, and is responsible for parsing and processing the received message and dispatching it to its destination via the messaging layer.
16.4 Details of the Pluggable-Protocol Framework

**Pluggable-Transport recv() Function**

**Synopsis**

```c
ssize_t recv (char* buffer,
              size_t len,
              const ACE_Time_Value* timeout = 0)
```

This function is the compliment to `send()`. It transfers a specified number of bytes from the transport layer into the supplied buffer, as long as the time limit provided is sufficient. Here is a TAO/UIOP_Transport example:

```c
ssize_t
TAO_UIOP_Transport::recv (char* buf,
                          size_t len,
                          const ACE_Time_Value* max_wait_time)
{
    ssize_t n = this->connection_handler_->peer ().recv (buf,
                                                     len,
                                                     max_wait_time);

    // Most of the errors handling is common for
    // Now the message has been read
    if (n == -1 &&
        TAO_debug_level > 4 &&
        errno != ETIME) {
        ACE_DEBUG ((LM_DEBUG,
                    ACE_TEXT ("TAO (%P|%t) - %p 
"),
                    ACE_TEXT ("TAO - read message failure ")
                    ACE_TEXT ("recv () \n")));
    }

    // Error handling
    if (n == -1) {
        if (errno == EWOULDBLOCK)
            return 0;

        return -1;
    }

    return n;
}
```

**Pluggable-Transport register_handler() Function**

**Synopsis**

```c
int register_handler (void)
```
This function is first called as result of the connection establishment, and the resulting instantiation and activation of the protocol-specific connection handler. Handler registration is preceded by a call to `register_handler()` on the transport’s wait strategy by the connector. The actual wait strategy is selected using the `-ORBClientConnectionHandler` client strategy factory option (see 21.2.3). Most protocols use the base class implementation provided by `TAO_Transport`. Some such as DIOP, HTIO etc. override this to either prevent handler registration or provide specialized implementations.

**Pluggable-Transport messaging_object() Function**

**Synopsis**

```
TAO_Pluggable_Messaging* messaging_object (void)
```

This function returns the messaging object that is used to format the data that needs to be sent and parse the data that is being received. Because the messaging object is instantiated in the protocol-specific transport constructor and stored in one of its member variables, this function is implemented in the concrete transport class.

**Pluggable-Transport event_handler_i() Function**

**Synopsis**

```
ACE_Event_Handler* event_handler_i (void)
```

This function returns the event handler used to receive notifications from the reactor.

**Pluggable-Transport connection_handler_i() Functions**

**Synopsis**

```
TAO_Connection_Handler * connection_handler_i (void)
```

This function returns the connection handler object associated with this transport.

**Pluggable-Transport messaging_init() Function**

**Synopsis**

```
int messaging_init(CORBA::Octet major, CORBA::Octet minor)
```

On the client side, this function is invoked by the invocation object. Its purpose is to forward the major and minor version numbers obtained from the protocol-specific profile object on to the messaging object. All protocols provided with TAO implement this code in the same way, and your custom protocol should do the same.
16.4 Details of the Pluggable-Protocol Framework

16.4.7.2 Transports in Use

To understand how the transport class functions during an invocation, it is helpful to have an overview of what happens to an invocation from beginning to end. This discussion is much like what we did earlier with the Messenger example, but with more detailed attention to the interactions that occur with the protocol-specific transport class.

First, we consider what happens on the client side when an invocation is made, as in the following code:

```c++
msg->send_message( "TAO User", "TAO Test", message inout() );
```

The stub code first instantiates a `TAO::Invocation_Adapter`:

```c++
TAO::Invocation_Adapter _tao_call (this, _the_tao_operation_signature, 1, "send_message", 4, this->the_TAO_MyTest_Proxy_Broker_);
```

This invocation adapter is then told to `invoke()`, thus establishing a connection to the remote server, and initializing the GIOP headers in the output CDR:

```c++
_tao_call.invoke (0, 0 ACE_ENV_ARG_PARAMETER);
```

In the case of a synchronous two-way invocation, the adapter delegates this to `TAO::Synch_Twoway_Invocation::remote_twoway()`.

Included with this connection establishment is the instantiation of a protocol-specific connection handler and transport. This process is set into motion through the use of the Connector pattern in conjunction with the Template Method pattern as described in 16.4.2.1:

```c++
template <class SVC_HANDLER> int TAO_Connect_Creation_Strategy<SVC_HANDLER>::make_svc_handler (SVC_HANDLER *&sh) {
    if (sh == 0)
        ACE_NEW_RETURN (sh, SVC_HANDLER (this->orb_core_, this->lite_flag_),
```
This is the exact location where the protocol-specific connection handler (and thus transport) is instantiated.

During connection establishment, a query is made on the transport for the major and minor version. These values are inserted into the output CDR stream that is transmitted to the server. The transport object delegates the retrieval of the version numbers to the messaging layer. A GIOP example of this is shown below:

```c
const TAO_GIOP_Message_Version& version = this->profile_->version ();
result = this->transport_->messaging_init (version.major,
                        version.minor);
```

The invocation then calls `generate_request_header()` to create the request header, which it delegates partially to the transport object.

Next the invocation object calls helper function `Remote_Invocation::marshal_data()` to marshall parameters into an output CDR stream.

Finally, the invocation calls base class function `Remote_Invocation::send_message()` which causes `send_message()` to be invoked on the transport object. It is inside this function that the fun begins with regard to the transport class.

Before actually sending the data, the transport object collaborates with the messaging layer to properly format the message inside the `OutputCDR` stream (e.g., adding the length field to the GIOP message):

```c
// Format the message in the stream first
if (this->messaging_object_->format_message (stream) != 0)
    return -1;
// "this" here is the pointer to the transport object.
```

After ensuring that the connection handler is registered with the reactor (to handle the reply when it arrives from the server) and formatting the message, the code in the protocol-specific transport then invokes the non-virtual `send_message_shared_i()` function in the `TAO_Transport` base class, which establishes the template by which the data is properly sent to the server.
Some of the actions dictated by this template are deferred to the protocol-specific derived-class transport, and some use implementations of both non-virtual and virtual functions in the base class. This is the canonical Template Method pattern in action.

It is at this point that the data being transferred changes its form from an `OutputCDR` stream into an `ACE Message_Block` chain, and ultimately into an `iovec` list for handling by the lower-level ACE classes.

The main template/algorithm for the send is as follows:

```c
int
TAO_Transport::send_message_shared_i (TAO_Stub* stub,
       int is_synchronous,
       const ACE_Message_Block* message_block,
       ACE_Time_Value* max_wait_time)
{
    if (message_semantics == TAO_Transport::TAO_TWOWAY_REQUEST)
    {
        return this->send_synchronous_message_i (message_block,
                                                max_wait_time);
    }
    else if (message_semantics == TAO_Transport::TAO_REPLY)
    {
        return this->send_reply_message_i (message_block,
                                                max_wait_time);
    }

    // Let's figure out if the message should be queued without trying
    // to send first:
    int try_sending_first = 1;
    int queue_empty = (this->head_ == 0);

    if (!queue_empty)
        try_sending_first = 0;
    else if (stub->sync_strategy ().must_queue (queue_empty))
        try_sending_first = 0;

    ssize_t n;

    TAO_Flushing_Strategy *flushing_strategy =
        this->orb_core ()->flushing_strategy ();

    if (try_sending_first)
    {
        size_t byte_count = 0;
        // ... in this case we must try to send the message first ...
```
size_t total_length = message_block->total_length();
if (TAO_debug_level > 6)
{
    ACE_DEBUG ((LM_DEBUG,
        "TAO (%P|%t) - Transport[%d]::send_message_i, "
        "trying to send the message (ml = %d)\n",
        this->id (), total_length));
}

// @@ I don't think we want to hold the mutex here, however if
// we release it we need to recheck the status of the transport
// after we return... once I understand the final form for this
// code I will re-visit this decision
n = this->send_message_block_chain_i (message_block,
    byte_count,
    max_wait_time);

if (n == -1)
{
    // ... if this is just an EWOULDBLOCK we must schedule the
    // message for later, if it is ETIME we still have to send
    // the complete message, because cutting off the message at
    // this point destroys the synchronization with the
    // server ...
    if (errno != EWOULDBLOCK && errno != ETIME)
    {
        if (TAO_debug_level > 0)
        {
            ACE_DEBUG ((LM_DEBUG,
                "TAO (%P|%t) - Transport[%d]::send_message_i, "
                "fatal error in "
                "send_message_block_chain_i %p\n",
                this->id (), ""));
            return -1;
        }
    }
}

// ... let's figure out if the complete message was sent ...
if (total_length == byte_count)
{
    // Done, just return. Notice that there are no allocations
    // or copies up to this point (though some fancy calling
    // back and forth).
    // This is the common case for the critical path, it should
    // be fast.
    return 0;
}

if (TAO_debug_level > 6)
{

16.4 Details of the Pluggable-Protocol Framework

ACE_DEBUG ((LM_DEBUG,
 "TAO (%P|%t) - Transport[%d]::send_message_i, "
 "partial send %d / %d bytes\n",
 this->id (), byte_count, total_length));

// ... part of the data was sent, need to figure out what piece
// of the message block chain must be queued ...
while (message_block != 0 && message_block->length () == 0)
 message_block = message_block->cont ();

// ... at least some portion of the message block chain should
// remain ...
ACE_ASSERT (message_block != 0);

// ... either the message must be queued or we need to queue it
// because it was not completely sent out ...

if (TAO_debug_level > 6)
{
 ACE_DEBUG ((LM_DEBUG,
 "TAO (%P|%t) - Transport[%d]::send_message_sharedi, "
 "message is queued\n",
 this->id ()));
}

if (this->queue_message_i(message_block) == -1)
{
 ACE_DEBUG ((LM_DEBUG,
 "TAO (%P|%t) - Transport[%d]::send_message_shared_i, "
 "cannot queue message for "
 " - %m\n",
 this->id ()));
 return -1;
}

// ... if the queue is full we need to activate the output on the
// queue ...
int must_flush = 0;
const int constraints_reached =
 this->check_buffering_constraints_i (stub,
 must_flush);

// ... but we also want to activate it if the message was partially
// sent.... Plus, when we use the blocking flushing strategy the
// queue is flushed as a side-effect of 'schedule_output()'

if (constraints_reached || try_sending_first)
{
(void) flushing_strategy->schedule_output (this);
}

if (must_flush)
{
    typedef ACE_Reverse_Lock<ACE_Lock> TAO_REVERSE_LOCK;
    TAO_REVERSE_LOCK reverse (*this->handler_lock_);
    ACE_GUARD_RETURN (TAO_REVERSE_LOCK, ace_mon, reverse, -1);

    (void) flushing_strategy->flush_transport (this);
}

return 0;
}

For normal synchronous messages, only the first few lines of the above
function are executed, in particular send_synchronous_message_i(). It is
a base-class non-virtual function, and imposes its implementation upon the
derived class as shown below:

int
TAO_Transport::send_synchronous_message_i (const ACE_Message_Block* mb,
                                       ACE_Time_Value* max_wait_time)
{
    // We are going to block, so there is no need to clone
    // the message block.
    TAO_Synch_Queue_Message synch_message (mb);
    synch_message.push_back (this->head_, this->tail_);

    int n =
        this->send_synch_message_helper_i (synch_message,
                                           max_wait_time);

    if (n == -1 || n == 1)
        return n;

    ACE_ASSERT (n == 0);

    TAO_Flushing_Strategy *flushing_strategy =
        this->orb_core ()->flushing_strategy ();
    (void) flushing_strategy->schedule_output (this);

    // Release the mutex, other threads may modify the queue as we
    // block for a long time writing out data.
    int result;
    {
        typedef ACE_Reverse_Lock<ACE_Lock> TAO_REVERSE_LOCK;
16.4 Details of the Pluggable-Protocol Framework

TAO_REVERSE_LOCK reverse (*this->handler_lock_);
ACE_GUARD_RETURN (TAO_REVERSE_LOCK,
    ace_mon,
    reverse,
    -1);

result = flushing_strategy->flush_message (this,
    &synch_message,
    max_wait_time);
}
if (result == -1)
{
    synch_message.remove_from_list (this->head_, this->tail_);
    if (errno == ETIME)
    {
        if (this->head_ == &synch_message)
        {
            synch_message.remove_from_list (this->head_, this->tail_);
            TAO_Queue_Message *queued_message = 0;
            ACE_NEW_RETURN (queued_message,
                TAO_Asynch_Queue_Message(
                    synch_message.current_block ( ),
                    0,
                    1),
                -1);
            queued_message->push_front (this->head_, this->tail_);
        }
    }
    if (TAO_debug_level > 0)
    {
        ACE_ERROR ((LM_ERROR,
            "TAO (%P|%t) - Transport[%d]:send_synchronous_message_i, ", 
            "error while flushing message %p\n",
            this->id (), ")");
    }
    return -1;
}
else
{
    ACE_ASSERT (synch_message.all_data_sent () != 0);
}

ACE_ASSERT (synch_message.next () == 0);
ACE_ASSERT (synch_message.prev () == 0);

return 1;
The above function puts the message block into a message object that is queued in the outgoing message queue. Once the message gets queued, the queue is drained. It is at this point that the data transforms from an \texttt{ACE\_Message\_Block} chain to an \texttt{iovec} list.

The data is then sent to the server in a protocol-specific way as shown below:

```c
ssize_t
TAO\_IIOP\_Transport::send (iovec* iov, int iovcnt,
    size_t& bytes\_transferred,
    const ACE\_Time\_Value* max\_wait\_time)
{
    ssize_t retval = this->connection\_handler\_peer ().sendv (iov, iovcnt,
        max\_wait\_time);

    if (retval > 0)
        bytes\_transferred = retval;
    else
    {
        if (TAO\_debug\_level > 4)
        {
            ACE\_DEBUG ((LM\_DEBUG,
                ACE\_TEXT ("TAO (%P|%t) - IIOP\_Transport[%d]:send, ")
                ACE\_TEXT ("send failure - %m (%d)\n"),
                this->id (), errno));
        }
    }

    return retval;
}
```

Once the invocation has been sent, the next step for the normal synchronous case is for the \texttt{Synch\_Two\_way\_Invocation} object to drive the \texttt{handle\_events()} event loop of the core reactor, so that the reply can be received from the server as shown below:

```c
const int reply\_error =
    this->resolver\_transport ()\->wait\_strategy ()\->wait (max\_wait\_time,
    rd);
```

At this point, the client blocks until the reply is received from the server. Once the reply comes back to the client side, the reactor-framework action kicks in and the \texttt{handle\_input()} function of the protocol-specific handler is called. The handler calls the \texttt{handle\_input\_internal()} base class, which delegates the receive action to the protocol-specific transport class as follows:
Details of the Pluggable-Protocol Framework

```c
int TAO_Connection_Handler::handle_input_internal (ACE_HANDLE h, ACE_Event_Handler * eh)
{
    // Let the transport know that it is used
    (void) this->transport ()->update_transport ();

    // Grab the transport id now and use the cached value for printing
    // since the transport could disappear by the time the thread
    // returns.
    size_t t_id =
        this->transport ()->id ();

    if (TAO_debug_level > 6)
    {
        ACE_HANDLE handle = eh->get_handle();
        ACE_DEBUG ((LM_DEBUG,
                    "TAO (%P|%t) - Connection_Handler[%d]:handle_input, 
                    "handle = %d/%d\n",
                    t_id, handle, h));
    }

    TAO_Resume_Handle resume_handle (this->orb_core (),
                                   eh->get_handle ());

    int return_value = 0;

    this->pre_io_hook (return_value);
    if (return_value != 0)
        return return_value;

    return_value =
        this->transport ()->handle_input (resume_handle);

    this->pos_io_hook (return_value);

    if (TAO_debug_level > 6)
    {
        ACE_HANDLE handle = eh->get_handle ();
        ACE_DEBUG ((LM_DEBUG,
                    "TAO (%P|%t) - Connection_Handler[%d]:handle_input, 
                    "handle = %d/%d, retval = %d\n",
                    t_id, handle, h, return_value));
    }

    return return_value;
}
```
The call to the transport’s handle_input() function in the above code is a call that is implemented by the TAO_Transport base class. It establishes another template (as in the Template Method pattern) for receiving data, just like send_message_shared() did for the send. Here is the implementation of handle_input():

```c++
int TAO_Transport::handle_input (TAO_Resume_Handle &rh,
                                 ACE_Time_Value * max_wait_time,
                                 int /*block*/) {
  if (TAO_debug_level > 3) {
    ACE_DEBUG ((LM_DEBUG,
        "TAO (%P|%t) - Transport[%d]:handle_input\n",
        this->id ())) ;
  }

  // First try to process messages of the head of the incoming queue.
  int retval = this->process_queue_head (rh);
  if (retval <= 0) {
    if (retval == -1) {
      if (TAO_debug_level > 2) {
        ACE_DEBUG ((LM_DEBUG,
            "TAO (%P|%t) - Transport[%d]:handle_input, "
            "error while parsing the head of the queue\n",
            this->id ())) ;
      }
      return retval;
    }
    if (TAO_debug_level > 2) {
      ACE_DEBUG ((LM_DEBUG,
          "TAO (%P|%t) - Transport[%d]:handle_input, "
          "error while parsing the head of the queue\n",
          this->id ())) ;
    }
  }

  return retval;
}

// If there are no messages then we can go ahead to read from the
// handle for further reading..

// The buffer on the stack which will be used to hold the input
// messages
char buf [TAO_MAXBUFSIZE];

#if defined (ACE_INITIALIZE_MEMORY_BEFORE_USE)
(void) ACE_OS::memset (buf,
  \'\0\',
  sizeof buf);
#endif
```
16.4 Details of the Pluggable-Protocol Framework

```c
#ifdef /* ACE_INITIALIZE_MEMORY_BEFORE_USE */

// Create a data block
ACE_Data_Block db (sizeof (buf),
    ACE_Message_Block::MB_DATA,
    buf,
    this->orb_core_->input_cdr_buffer_allocator (),
    this->orb_core_->locking_strategy (),
    ACE_Message_Block::DONT_DELETE,
    this->orb_core_->input_cdr_dblock_allocator ());

// Create a message block
ACE_Message_Block message_block (&db,
    ACE_Message_Block::DONT_DELETE,
    this->orb_core_->input_cdr_msgblock_allocator ());

// Align the message block
ACE_CDR::mb_align (&message_block);

size_t recv_size = 0;

if (this->orb_core_->orb_params ()->single_read_optimization ())
{
    recv_size =
        message_block.space ();
}
else
{
    recv_size =
        this->messaging_object ()->header_length ();
}

// If we have a partial message, copy it into our message block
// and clear out the partial message.
if (this->partial_message_ != 0 && this->partial_message_->length () != 0)
{
    if (message_block.copy (this->partial_message_->rd_ptr (),
        this->partial_message_->length ()) == 0)
    {
        recv_size -= this->partial_message_->length ();
        this->partial_message_->reset ();
    }
    else
    {
        ACE_ERROR_RETURN ((LM_ERROR,
            "TAO (%P|%t) - Transport[%d]:handle_input, 
            "unable to copy the partial message\n",
            this->id ()),
                -1);
    }
```
Developing Pluggable Protocols

this->recv_buffer_size_ = recv_size;

// Read the message into the message block that we have created on
// the stack.
ssize_t n = this->recv (message_block.wr_ptr (),
    recv_size,
    max_wait_time);

// If there is an error return to the reactor..
if (n <= 0)
{
    return n;
}

if (TAO_debug_level > 2)
{
    ACE_DEBUG ((LM_DEBUG,
        "TAO (%P|%t) - Transport[%d]::handle_input, "
        "read %d bytes
",
        this->id (), n));
}

// Set the write pointer in the stack buffer
message_block.wr_ptr (n);

// Parse the message and try consolidating the message if
// needed.
retval = this->parse_consolidate_messages (message_block,
    rh,
    max_wait_time);

if (retval <= 0)
{
    if (retval == -1 && TAO_debug_level > 0)
    {
        ACE_DEBUG ((LM_DEBUG,
            "TAO (%P|%t) - Transport[%d]::handle_input, "
            "error while parsing and consolidating\n",
            this->id ()));
    }
    return retval;
}
16.4 Details of the Pluggable-Protocol Framework

As with the `send_message_shared_i()` function, some function calls are deferred to the protocol-specific derived transport class to execute the protocol-specific behavior. Once the code that is common to all transport classes has finished executing, a call is made to the transport-specific `recv()` as shown below:

```c
ssize_t TAO_IIOP_Transport::recv (char* buf,
   size_t len,
   const ACE_Time_Value* max_wait_time)
{
   ssize_t n = this->connection_handler_->peer ()->recv (buf,
               len,
               max_wait_time);

   // Do not print the error message if it is a timeout, which could
   // occur in thread-per-connection.
   if (n == -1 &&
        TAO_debug_level > 4 &&
        errno != ETIME) {
      ACE_DEBUG ((LM_DEBUG,
```
Developing Pluggable Protocols

Here is where the protocol-specific transport finally makes use of specific ACE classes to receive the incoming data.

After the `recv()` has finished executing within the `handle_input()` function, calls are made to non-virtual `TAO_Transport` functions which perform protocol-independent actions that get the reply where it is supposed to go. This code also re-registers the connection handler with the reactor, so that it can be used for future replies.

The normal synchronous request is the simplest case. Things get more complicated with asynchronous requests, oneways, and buffered versions of these two. But the basic scheme is the same. You do not have to worry about much of the transport code, because it is taken care of in the transport base class. As you can tell from the code above, and also if you examine the code for existing protocols, most of the functionality is performed by the base class, with only a few protocol-specific actions deferred to the derived class.

The server side is essentially a mirror image of the client side with regard to the transport. The connection-handler and transport classes are the same and the same Template Method pattern functions are called on both sides to achieve different functionality. Whether the `handle_input()` template function is called on the server side as a result of a request arrival, or whether it is called on the client side as the result of a reply received from the server, its execution is very similar. Likewise, whether the `send_message_shared()` template function is called on the client side as
result of a client invocation going out, or on the server side as result of reply being sent back to the client, its behavior is quite similar in both client and server.

Each of these template functions defers protocol-specific functionality to concrete derived transport classes.

### 16.4.8 Using ACE_wrappers

As you can see from the source code of the protocols provided with TAO, they leverage many existing ACE classes to perform key protocol-specific actions. In fact, in all the protocols provided with TAO, the actual sending and receiving of data is delegated to these protocol-specific ACE classes.

By far, the easiest way to integrate your new transport protocol into the TAO pluggable-protocol framework is to begin by creating these protocol-specific classes based on the ACE Inter-Process Communication Service Access Point (IPC_SAP) family of classes, described in *C++ Network Programming, Volume 1: Mastering Complexity Using ACE and Patterns* (C++NPv1) (2001) by Doug Schmidt and Steve Huston.

Doing so allows you to capitalize on the existing support code provided for you in the ACE framework. For example, using the ACE framework allows you to seamlessly integrate your classes with the ACE reactor mechanisms and the various ACE strategy-based behavior controls. Using the ACE classes and templates as the basis for your pluggable protocol significantly reduces the effort needed to integrate it with TAO. In fact, it becomes nearly a cut-and-paste operation.

Within the **IPC_SAP** family of classes are two root classes from which the related types are derived. They are:

- **ACE_IPC_SAP**, which supplies a handle used for registration with the reactor. Any class that requires such a handle must derive from **ACE_IPC_SAP**. Typically, for a communication mechanism such as sockets or pipes, a base type is derived from **ACE_IPC_SAP** that is then specialized into such things as streams and acceptors. Because connectors do not need to register with the reactor, they usually do not need to inherit from **ACE_IPC_SAP**.
- **ACE_Addr**, the base type for all endpoint-definition classes.
Derive your protocol interface from these ACE classes to obtain a substantial reduction in the effort required to build a new TAO pluggable protocol.

16.5 Pluggable Protocol Summary

This chapter has presented an overview of how protocols are used by TAO, a description of what the TAO internals expect of your custom protocol, and the exact details of how to implement your custom protocol. Of course, each protocol type presents its own special issues and challenges that can be substantially different from what has been presented here. Even in these cases, you should now have an intuitive understanding of the expectations that TAO places on pluggable protocols, and how to fulfill these expectations.
CHAPTER 17

Multithreading with TAO

17.1 Introduction

TAO is used in many application domains, across a wide variety of platforms and operating systems, and often subject to wide-ranging performance requirements. Often, overall throughput of an application can be improved by using multithreaded programming techniques. For example, multithreading can be useful in the following situations:

- The application is characterized by long-running operations that are largely independent of one another.
- The application must support a large number of clients whose requests must be executed concurrently.
- Data and resource sharing among operations is minimal and easily identifiable.
- Operations can be executed in parallel.
- Servant implementations are thread-safe (reentrant) (i.e., can be safely executed concurrently by different threads) and appropriate
synchronization mechanisms (e.g., mutex locks) are used to protect critical regions of code from concurrent execution.

- Legacy and third-party code and libraries used by servant implementations are also thread-safe; or if they are not they are protected from concurrent execution.

**Note** This chapter assumes the reader is somewhat familiar with the fundamental concepts of multithreaded programming. For more information, see Chapter 5 of *C++ Network Programming: Volume 1 (C++NPv1)* and Chapter 2 of *Threads Primer*.

### 17.1.1 Road Map

This chapter describes several aspects of multithreading for TAO applications, including application programming interfaces (both CORBA-compliant and TAO-specific), policies, configuration options, and design choices. While the chapter is intended to be read in its entirety, you may find benefit in reading certain sections independently.

- Section 17.2, “Overview of Client/Server Roles in CORBA,” defines the client and server roles in CORBA and reviews the stages of request invocation by clients and request processing by servers.

- Section 17.3, “Multithreading in the Server,” describes configuration and behavior for a thread in the server role.
  - Section 17.3.1, “Event Handling,” and Section 17.3.2, “Shutdown and Destruction,” describe ORB operations for providing one or more threads to the ORB, shutting down the ORB, and destroying the ORB. TAO’s ORB supports the thread-related operations defined by the CORBA specification and provides certain TAO-specific extensions.
  - Section 17.3.3, “Request Processing,” describes how TAO chooses a thread for dispatching a request and describes the POA threading models and associated policy settings that affect concurrent processing of requests. It also describes the various server concurrency models provided by TAO and the options and mechanisms to configure them, including the motivating factors for each concurrency model and the consequences of using each one.
This section includes examples of applications using the various concurrency models.

- Section 17.4, “Multithreading in the Client,” describes configuration and behavior for a thread in the client role. We provide guidelines for configuring TAO clients to achieve intended behavior while avoiding common pitfalls that can arise from incorrectly applying TAO’s concurrency-related configuration options.
  - Section 17.4.1, “Establishing a Connection to the Server,” describes TAO’s connection strategies, by which a TAO client will wait for a connection to a server to be established.
  - Section 17.4.2, “Multiplexing Requests on a Connection,” describes how a single connection can be used for multiple simultaneous requests.
  - Section 17.4.3, “Flushing Requests to the Server,” describes TAO’s flushing strategies, by which a TAO client thread will write request messages to the transport layer.
  - Section 17.4.4, “Waiting for a Reply from the Server,” describes TAO’s wait strategies, the client-side concurrency strategies affecting both how a client waits for a reply to come from a server and what types of activities it might do in the meantime.
  - Section 17.4.5, “Optimizing Performance of Collocated Objects,” describes TAO’s collocation optimizations, through which a client can configure trade-offs between performance and behavior for collocated objects.

17.2 Overview of Client/Server Roles in CORBA

Usually, a discussion of multithreading with respect to CORBA applications is oriented toward server-side behavior. Indeed, CORBA servers typically benefit most from the use of multithreading techniques. However, some multithreading issues are relevant to client behavior as well.

In practice, few CORBA applications are either pure clients (making only outbound requests and receiving replies) or pure servers (only processing inbound requests and returning replies). Hybrid applications that play the role of both server and client, depending upon the semantics of a particular task or
object interaction, are typical. Therefore, in many places in this chapter, to
clarify the discussion of TAO’s strategies associated with client/server
behavior, the following definitions apply:

- A **client thread** is a thread playing the role of a client; i.e., invoking a
  request, waiting for a reply.
- A **server thread** is a thread playing the role of a server; i.e., available to
  the ORB to handle connections from clients, process inbound requests,
  send replies, etc.

Note that a given thread can switch between these two roles. For example,
while processing an inbound request, a server thread may invoke an outbound
request on another CORBA object, thereby switching to the client role. It may
remain in the client role until the reply to that outbound request is received. It
then switches back to the server role to prepare and send the reply to the
original inbound request.

Likewise, under certain conditions a client thread may temporarily switch to
the server role. For example, while waiting for a reply to an outbound request,
a thread may be made available to the ORB for processing incoming client
connections and requests, then return to the client role to handle its reply when
it arrives.

It is important to note that by invoking a request on another CORBA object,
the application controls when a thread switches from the server role to the
client role. However, the configuration of the ORB at run time influences
whether a thread’s role can change in the opposite direction, from the client
role to the server role. It may only happen under certain conditions, such as
when a single-threaded hybrid application must respond to client connections
and requests even while it is waiting for a reply to an outbound request.

### 17.2.1 The Client’s Role: Overview of Request Invocation

Invoking a synchronous CORBA request can be viewed as a four-step
process:

1. *A client thread invokes an operation defined in the target object’s IDL
   interface.*
   The client invokes the operation on a client-side proxy or stub. The proxy
   prepares a CORBA request message to send to the target object. For our
   purposes, we assume the target object is located in a different address
   space (and possibly on a different host) than the client.
2. The client-side ORB sends the request message to the target object.
   The client requires a connection to transmit the request message. The client-side ORB may already have established a connection to the server. If not, a connection must be established to transmit the client’s request.

3. The client waits for the reply.
   How the client thread waits for the reply depends upon the wait strategy configured at run-time. In some cases, the thread may be made available for handling inbound CORBA requests or other events until its reply is received. If so, the thread may temporarily switch to the server role.

4. The client receives and processes the reply.
   The client-side ORB receives the reply message sent from the server and returns it to the proxy. The proxy demarshals the reply and returns to the client application code that initially invoked the operation. The client thread then continues.

The steps above assume the invocation was a synchronous (two-way) request on a remote CORBA object. Different steps may be followed in the case of either a oneway request, an Asynchronous Method Invocation (AMI), or an invocation on a collocated object. Policies that affect invocation behavior for oneway requests, as well as the technique of AMI, are described in Chapter 7. Strategies for controlling thread behavior for invocations on collocated objects are discussed later in this chapter in 17.4.5.

17.2.2 The Server’s Role: Overview of Request Processing
Processing a CORBA request can be generalized as a five-step process:

1. The ORB binds the request to a thread.
   A request message sent to a server remains queued in the transport’s input buffer until a thread is available to process it. How the ORB selects a thread to process the request depends upon several factors, such as how the application developer has chosen to make threads available to the ORB and strategies configured at run-time that affect threading behavior. Often, the same thread is used throughout the processing of the request.

2. The ORB dispatches the request to a POA.
   After selecting a thread to process the request, the ORB identifies the target POA and invokes the POA to process the request.

3. The POA dispatches the request to an application object (servant).
To dispatch a request to a servant, a POA applies a threading model to enforce constraints on concurrent processing and a request processing policy to identify and invoke the target object, or servant. In addition, some POAs may employ customized servant dispatching strategies. The application developer chooses the threading model and request processing policy used by the POA.

4. The servant executes and returns.

The server thread executes application code to satisfy the request. Depending upon the application, this code may switch to the client role and invoke outbound requests on other CORBA objects. If so, it then must wait for its reply, in which case it may switch back (temporarily) to the server role, as described in 17.2.1. Eventually, the request that was invoked on this servant is satisfied and the application code returns.

5. The server-side ORB creates a reply message and sends it to the client.

In the case of a synchronous (two-way) request, the return from the servant causes a reply message to be generated and sent back to the client.

The CORBA Core specification (OMG Document formal/04-03-12) defines operations to enable CORBA servers to process requests. For example, a server thread calls `CORBA::ORB::run()` to run the ORB’s event loop, thus making that thread available to the ORB for dispatching requests. In addition, the specification defines POA policies that applications can use to enforce constraints on concurrent request processing and control how the POA chooses the target object for each request.

Often, the operations and policies defined in the CORBA specification are insufficient for application developers to effectively control and manage the request processing behavior of their CORBA applications. For example:

- A server may need to provide multiple threads to efficiently process incoming requests, perhaps from multiple clients, in a timely manner.
- A single-threaded server that also acts in a client role may need to be able to respond to incoming requests while it is waiting for a reply to an outbound request to avoid a possible deadlock situation.
- A middle-tier server in a three-tier architecture may be used to break up large blocks of incoming data into smaller blocks that can be processed independently. This middle-tier may delegate processing of these smaller blocks of data to back-end servers, gather together the individual results...
from the back-end servers, then formulate and return a single reply to the originating client.

- A server may need to de-couple threads allocated to receiving client requests from threads used to process and reply to such requests.

In fact, the CORBA specification gives ORB implementers a great deal of latitude in how they deal with threading and concurrency issues. Therefore, TAO provides several strategies that applications can use to control concurrency and request processing by servers. Some of these strategies can be completely configured at run time; others require use of specific APIs in code. We discuss several such strategies later in this chapter.

In addition, TAO provides a technique called *Asynchronous Method Handling* (AMH) that allows an application to delegate processing of a synchronous request and transmission of a reply to a thread other than the one from which the servant’s method was invoked. For more information on Asynchronous Method Handling, please see Chapter 8.

### 17.3 Multithreading in the Server

The server’s role in a distributed application is a passive one, as the server waits for requests from clients. The server-side ORB has several operations to control the server’s handling of requests from clients. The ORB’s thread-related operations are used to do the following:

- Provide one or more threads to an ORB to handle events.
- Shut down an ORB.
- Destroy an ORB.

The relevant portions of the ORB’s interface definition are:

```idl
// IDL
module CORBA
{
    // Portions of the CORBA module omitted.

    interface ORB
    {
        // Portions of the ORB interface omitted.

        void run();
    }
}
```
The relevant operations can be summarized as follows:

- **CORBA::ORB::run()** provides a thread to the ORB to handle events. In appropriate circumstances, this operation can be called from multiple threads to provide more than one thread to the ORB.

- **CORBA::ORB::perform_work()** provides a thread to the ORB for a single unit of work. The definition for a unit of work is left to ORB implementers; often it is a single invocation. The perform_work() operation is best used in conjunction with CORBA::ORB::work_pending() to implement a polling loop that interleaves ORB processing with other activities.

- **CORBA::ORB::shutdown()** instructs an ORB to halt processing. This operation is typically invoked just prior to the ORB’s destruction.

- **CORBA::ORB::destroy()** destroys an ORB and releases its resources so they can be reclaimed by the application.

The remainder of the section elaborates on each operation.

### 17.3.1 Event Handling

There are two CORBA-compliant operations for handling requests from clients. The **CORBA::ORB::run()** operation provides a self-contained CORBA event loop. The **CORBA::ORB::perform_work()** operation is designed to act as part of a larger event loop that interleaves non-CORBA behavior with a CORBA event loop. Both of these operations are CORBA-compliant, but provide TAO-specific extensions.

In addition, a TAO server can interleave non-CORBA behavior with CORBA events by registering an event handler with TAO’s Reactor. Accessing TAO’s Reactor is discussed in more depth in 17.3.1.3. Further information on the Reactor is available in Chapter 3 of *C++ Network Programming: Volume 2 (C++NPv2)* by Douglas C. Schmidt and Stephen D. Huston.
17.3 Multithreading in the Server

17.3.1.1 CORBA::ORB::run()

CORBA::ORB::run() provides a thread to the ORB to handle events. In appropriate circumstances, this operation can be called from multiple threads to provide more than one thread to the ORB. This is a blocking operation from the thread’s perspective; each thread that calls run() becomes dedicated to the ORB. The run() operation does not return until the ORB has shut down. More information on providing multiple threads to the ORB is available in the discussion of the thread-pool concurrency model in 17.3.3.2.

TAO Extensions

TAO provides overloaded versions of CORBA::ORB::run() that accept an ACE_Time_Value indicating the maximum amount of time the ORB should run its event loop before returning. The declarations of these functions, in C++, are as follows (found in $TAO_ROOT/tao/ORB.h):

```cpp
// C++
namespace CORBA {
    class ORB {
        // Portions of the ORB class definition omitted.

        public:
            void run (ACE_Time_Value &tv);
            void run (ACE_Time_Value *tv);
    };
}
```

ACE_Time_Value, defined in $ACE_ROOT/ace/Time_Value.h, expresses time in seconds and microseconds. For more information on ACE_Time_Value, please see Chapter 3 of C++ Network Programming: Volume 2 (C++NPv2) by Douglas C. Schmidt and Stephen D. Huston.

If a time value is passed to the ORB’s run() method, it will block until the ORB has shut down or until it has completed processing events that are initiated within the specified time-out period. If run() returns before the time-out period has expired (e.g., due to ORB shut down, the process receives a signal, etc.), the value of the ACE_Time_Value parameter will have been reduced by the amount of time spent in the call. In the second form of the function above, a pointer value of 0 specifies an infinite time-out period.
Multithreading with TAO

Note Using these overloaded versions of CORBA::ORB::run() reduces the portability of your application code to other ORB implementations.

Note You should not use these overloaded versions of CORBA::ORB::run() in more than one thread unless each thread passes its own ACE Time_Value instance. If multiple threads call run() with the same time value instance, they may concurrently modify the time value instance, leading to unexpected behavior.

17.3.1.2 CORBA::ORB::perform_work()

CORBA::ORB::perform_work() provides a thread to the ORB for a single unit of work. This is a non-blocking operation from the calling thread’s perspective; the thread can perform tasks unrelated to the ORB between calls to perform_work(). The definition for a unit of work is left to ORB implementers; often it is a single invocation.

The perform_work() operation is best used in conjunction with CORBA::ORB::work_pending() to implement a polling loop that interleaves ORB processing with other activities, such as servicing another external interface. However, this can lead to jittery performance compared with calling CORBA::ORB::run() in its own thread. The boolean return value from work_pending() indicates whether or not an ORB has an immediate need for a thread to perform ORB-related activities.

TAO Extensions

TAO provides overloaded versions of CORBA::ORB::perform_work() that accept an ACE_Time_Value indicating the maximum amount of time the ORB should stay in perform_work() before returning. The declarations of these functions, in C++, are as follows (found in $TAO_ROOT/tao/ORB.h):

```cpp
namespace CORBA
{
    class ORB
    {
        // Portions of the ORB class definition omitted.

        public:
            void perform_work (ACE_Time_Value &tv);
            void perform_work (ACE_Time_Value *tv);

```
17.3 Multithreading in the Server

Note In TAO, when CORBA::ORB::perform_work() is called without a time value, the ORB may handle multiple events before returning. These events may include input events (such as inbound requests or replies), output events (such as outbound requests or replies), timers, or signals. There is no guarantee that perform_work() will return after dispatching just one event.

Note The CORBA specification states that only the main thread should use work_pending() and perform_work(). There is no such restriction in TAO.

17.3.1.3 Guidelines for Application Developers

For an application to behave in the server role, at least one thread must be made available to the ORB by calling CORBA::ORB::run() or by periodically calling CORBA::ORB::perform_work(). A thread from which run() is called becomes dedicated to the ORB. A thread that calls perform_work() can be used to perform other tasks in addition to ORB-related tasks.

Typical CORBA server architectures include:

- A single-threaded server that has all of its activities initiated via CORBA invocations typically calls run() from the main thread after the application is initialized.
- A multithreaded server that has all of its activities initiated via CORBA invocations typically calls run() from each thread.
- A single-threaded server that has processing tasks unrelated to CORBA invocations has three options:
  1. Implement a polling loop, using work_pending() and perform_work(), to interleave CORBA invocations with other processing tasks. A typical polling loop is similar to the following:

     ```cpp
     for(;;)
     {
       if(orb->work_pending())
       {
     ```
In most cases, \texttt{perform\_work()} should be called only when \texttt{work\_pending()} returns \texttt{TRUE} to prevent the application from blocking in the absence of CORBA invocations. However, TAO allows time-bounded calls to \texttt{perform\_work()} and thus removes the need to call \texttt{work\_pending()}. This use of \texttt{perform\_work()} is shown in the following code fragment:

```c
for(;;)
{
    // wait for incoming requests for 100 milliseconds
    ACE_Time_Value tv(0, 100000); // 100 msec == 100000 usec
    orb->perform_work(tv);
    // do other tasks
}
```

This call to \texttt{perform\_work()} blocks until either a unit of work is processed or at least one second passes, whichever comes first. The time spent in \texttt{perform\_work()} is not deterministic, as is discussed below.

2. Implement a time-bounded \texttt{CORBA::ORB::run()} loop, using TAO’s \texttt{ACE\_Time\_Value\_based} extensions. For example:

```c
for(;;)
{
    // wait for incoming requests for 100 milliseconds
    ACE_Time_Value tv(0, 100000); // 100 msec == 100000 usec
    orb->run(tv);
    // do other tasks
}
```

This call to \texttt{run()} blocks until at least one second passes. The time spent in \texttt{run()} is not deterministic.

3. Register an event handler with TAO’s reactor and handle both ORB and non-CORBA events with \texttt{orb->run()}, thus interleaving ORB events with events from other sources. This is discussed in more detail below.
17.3 Multithreading in the Server

- A multithreaded server that has tasks unrelated to CORBA invocations has two options:

  1. Dedicate each thread to either performing ORB-related activities using `orb->run()`, or to other activities using the appropriate mechanism.

  2. Dedicate a group of threads to the ORB using `orb->run()` and use a polling loop to interleave ORB-related and other activities on another thread or group of threads.

  3. Register an event handler with TAO’s reactor and handle both ORB and non-CORBA events with multiple threads using `orb->run()`, thus interleaving ORB events with events from other sources. This is discussed in more detail below.

Whether or not to use multiple threads for activities unrelated to the ORB depends upon the nature of those activities and the applicable concurrency constraints. When a polling loop is used in multithreaded applications, it is often executed on the main thread after other threads that are dedicated to the ORB have been spawned.

The run time of `perform_work()` is not deterministic. Using `perform_work()` to interleave CORBA invocations with other processing activities can be tricky. Numerous factors, including long-running CORBA requests, outbound CORBA requests that are issued during the processing of an inbound CORBA request, timers, and signals influence the run time of `perform_work()`. As described above, TAO extends `perform_work()` to permit specification of a maximum run time, but typically this is not sufficient to place an absolute limit on `perform_work()`’s run time. For example, when a synchronous outbound request is issued during the processing of an inbound request, the run time for the inbound request becomes a function of the outbound request’s run time, which is beyond the scope of the local ORB.

**Interleaving ORB Events with Events from Other Sources**

Often, an application that plays the role of a CORBA server must also be responsive to events from other sources, such as a graphical user interface or a socket-based messaging or data distribution infrastructure. These applications may need to interleave ORB events (inbound client connections and requests, outbound requests and replies) with these non-CORBA events. As discussed above, there are several options for accomplishing this interleaving of events. For example, the application could:
• Use non-blocking ORB event handling operations (`work_pending()` and `perform_work()`) to implement a polling loop. See the discussion above for more information.

• Dedicate separate threads for CORBA request processing and for processing events from other sources. See the discussion above for more information.

• Register its own custom event handlers with the ORB’s reactor and handle both ORB and non-CORBA events with `orb->run()`. For example:

```cpp
// get TAO’s reactor
ACE_Reactor* reactor = orb->orb_core()->reactor();

// ... create an event handler, register it with the Reactor ...

// handle CORBA and non-CORBA events
orb->run();
```

More information on creating an event handler and registering it with the reactor is available in Chapter 3 of *C++ Network Programming: Volume 2 (C++NPv2)* and in Chapter 7 of *The ACE Programmer’s Guide (APG)*.

• Use one of the special resource factory and reactor types provided by TAO that are designed for integrating with other event sources. For example, see 20.3 for information on integrating TAO with the Qt GUI toolkit, 20.4 for information on integrating TAO with the X Window System XtIntrinsics toolkit, and 20.7.6 for information on using the WFMO reactor with TAO.

• Implement a custom reactor type that integrates CORBA events with events from other sources.

### 17.3.2 Shutdown and Destruction

The `CORBA::ORB::shutdown()` and `CORBA::ORB::destroy()` operations halt the ORB’s processing and release the ORB’s resources, respectively. Both of these operations are CORBA-compliant; neither has TAO-specific extensions. This section contains descriptions of both operations and guidelines for successful execution of shutdown and destruction.
17.3 Multithreading in the Server

17.3.2.1 CORBA::ORB::shutdown()

CORBA::ORB::shutdown() instructs an ORB to halt processing. This operation is typically invoked just prior to the ORB’s destruction. If the wait_for_completion parameter is TRUE, this operation blocks until the ORB has concluded request processing and other object adapter related activities, and all object adapters have been destroyed. If wait_for_completion is FALSE, then shutdown() may return before the ORB completes the process of shutting down.

Note

The ORB continues normal processing activities while it is shutting down, so there may be a significant delay after shutdown() is called before the ORB actually stops processing. For example, if shutdown() is called with wait_for_completion equal to TRUE, the ORB waits for all threads that are processing ORB events to exit before returning.

Exceptions

The following exceptions may be raised by the ORB’s thread-related operations or by circumstances arising from the use of those thread-related operations:

- work_pending() or perform_work() called on an ORB after it has been shut down raises CORBA::BAD_INV_ORDER.

- shutdown() with wait_for_completion equal to TRUE called from a thread processing a CORBA request raises CORBA::BAD_INV_ORDER with an OMG minor code of 3 and a completion status of COMPLETED_NO.

- Any operation other than duplicate(), release(), or isNil() invoked on an ORB after it has shut down, or invoked on an object reference obtained through an ORB that has since shut down, raises CORBA::BAD_INV_ORDER with an OMG minor code of 4. Scenarios may arise in which operations other than those cited here may be called while an ORB is shutting down. Application developers should be prepared to handle this exception when such scenarios may occur because the time required to shut down an ORB is not predictable. (See 17.3.2.3.)

17.3.2.2 CORBA::ORB::destroy()

CORBA::ORB::destroy() destroys an ORB and releases its resources so they can be reclaimed by the application. This operation initiates the shutdown
process when called on an ORB that has not been shut down. If destroy() initiates the shutdown process, then it blocks until the ORB has shut down as if shutdown() had been called with wait_for_completion equal to TRUE.

Exceptions
The following exceptions may be raised by the ORB’s thread-related operations or by circumstances arising from the use of those thread-related operations:

- Any operation invoked on an ORB after its destruction raises CORBA::OBJECT_NOT_EXIST.
- destroy() called from a thread processing an invocation raises CORBA::BAD_INV_ORDER with an OMG minor code of 3.

17.3.2.3 Guidelines for Application Developers
Terminating a CORBA server gracefully is often a complicated matter because the ORB continues normal processing while it shuts down. An application dedicated to processing requests can be signaled to shut down via a CORBA request. An application that performs activities unrelated to CORBA requests can be signaled to shut down via a CORBA request as well as any other interfaces that are serviced outside the scope of CORBA requests.

If signaled to shut down via a CORBA interface, shutdown() can be invoked with wait_for_completion equal to FALSE from the thread that processes the request; invoking shutdown() with wait_for_completion equal to TRUE would raise an exception in this situation. If signaled to shut down outside the scope of a CORBA request, the responding thread can invoke shutdown() with wait_for_completion equal to TRUE or FALSE; wait_for_completion equal to TRUE in this case will block the calling thread until the ORB has shut down.

Once shutdown() has been called, all calls to run() will return once the ORB has shut down. However, the time required to shut down an ORB depends upon the volume of client activity when shutdown is requested. In extreme cases, a complete cessation of client activity may be necessary to allow the ORB to shut down.

Here are some options, both graceful and ungraceful, for shutting down a CORBA-based application:
17.3 Multithreading in the Server

- The application can be killed with the appropriate platform-specific mechanism. This is the simplest solution but it is not graceful and may have undesirable side-effects.

- CORBA::ORB::shutdown() can be called with wait_for_completion equal to FALSE from a thread processing a CORBA request that was invoked via some maintenance or administrative interface. All calls to CORBA::ORB::run() return once the ORB has shut down and thus release the threads dedicated to the ORB. Some mechanism should be used to prevent the main thread from exiting before the other threads exit.

The application may continue to run for a relatively long time. If this is unacceptable, another mechanism should be used to forcibly terminate the application.

- The application can block the main thread on a condition variable or semaphore until the application is signaled to shut down. (In this case, the main thread is not used for ORB-related activities.) Once unblocked, the main thread calls CORBA::ORB::shutdown() to terminate ORB-related processing. If shutdown() is called with wait_for_completion equal to FALSE, some other mechanism should be used to prevent the main thread from exiting before all other threads have exited.

This option provides some additional flexibility. After the main thread initiates shutdown, it might then block until all other threads have exited or a specified time interval has elapsed. If the other threads exit before the time interval elapses, the main thread conducts any other cleanup activities and then exits. If the time period elapses, the main thread performs some, perhaps not all, of the usual clean-up activity and then exits causing a forced termination.

The next section contains more information on gracefully shutting down a TAO server.

Gracefully Shutting Down a TAO Server

In practice, CORBA applications are often designed for 24/7 operation. If a server is not running, the system is non-functional. Still, there are often situations in which a server needs to be gracefully shut down (e.g., to replace a running server with a newer version or to move a running server to a different host). By a graceful shutdown, we mean something slightly more elegant than, for example, sending the process a kill signal.
Gracefully shutting down a TAO server can be tricky. An application could do one of the following:

- **Use non-blocking ORB event handling operations** (`work_pending()` and `perform_work()`) to implement a polling loop as described in 17.3.1.3. An application-specific flag could be used to indicate when the loop should be terminated.

  The following code fragment shows how this could be done:

  ```cpp
  bool done = false;
  do {
    // wait for incoming requests for a maximum of 100 milliseconds
    ACE_Time_Value tv(0, 100000); // 100 msec == 100000 usec
    orb->perform_work(tv);
    // do other tasks
    if ( /* some exit condition exists */ )
      done = true;
  } while (!done);
  ```

- **Use** `CORBA::ORB::run()` **in one or more threads to process CORBA requests**, then invoke `CORBA::ORB::shutdown()` from a thread that is not being used to process CORBA requests. For example, the application could provide one thread that monitors console input for an application-specific command such as “quit.” If the `wait_for_completion` parameter to `shutdown()` is `TRUE`, request processing could continue for some time after the call the `shutdown()`.

- **Call** `CORBA::ORB::shutdown()` **within the context of processing a request**. For example, the application could implement an interface that has a `shutdown()` operation, such as the following:

  ```cpp
  interface MyAppAdmin
  {
    oneway void shutdown ();
  };
  ```

  One possible implementation of the `MyAppAdmin::shutdown()` operation could be as follows:

  ```cpp
  void MyAppAdmin_i::shutdown () throw (CORBA::SystemException)
  {
    // shut down the ORB (stored in a CORBA::ORB_var variable orb_)
  }
  ```
17.3 Multithreading in the Server

Note that the application passes FALSE for the wait_for_completion parameter to CORBA::ORB::shutdown() as required when shutdown() is called within the context of a request; otherwise, the ORB raises the CORBA::BAD_INV_ORDER system exception with an OMG minor code of 3.

- Register a timer (derived from ACE_Event_Handler) with the ORB’s reactor (or a separate reactor) and call CORBA::ORB::shutdown() in the timer’s handle_timeout() method.

The following code fragment shows how such a timer class could be implemented:

```cpp
class MyAppShutdownTimer : public ACE_Event_Handler
{
public:
    // Constructor
    MyAppShutdownTimer (CORBA::ORB_ptr orb)
        : orb_(CORBA::ORB::_duplicate(orb)) { }
    // Receive timeout events from the Reactor
    virtual int handle_timeout (const ACE_Time_Value &current_time,
                                 const void *act)
    {
        orb_->shutdown (0);
        return 0;
    }
private:
    CORBA::ORB_var orb_;  
};
```

The following code fragment shows how to create and schedule a timer with the ORB’s reactor:

```cpp
    // Portions of int main(int, char *argv[]) omitted.

    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Intervening code omitted.

    // Create a shutdown timer.
    MyAppShutdownTimer * timer = new MyAppShutdownTimer (orb.in());
```
Multithreading with TAO

// Schedule the timer to shutdown the ORB in 30 seconds.
ACE_Time_Value timeout (30,0);
orb->orb_core()->reactor()->schedule_timer(
    timer, 0, timeout);

// Run the ORB’s event loop.
orb->run();
orb->destroy();

• Register a signal handler (derived from ACE_Event_Handler) with the ORB’s reactor (or a separate reactor) and call CORBA::ORB::shutdown() in the timer’s handle_signal() method.

• Use the overloaded version of CORBA::ORB::run() or CORBA::ORB::perform_work() that processes events for a specified time interval.

An example showing many of the above techniques for gracefully shutting down a TAO server can be found in the $TAO_ROOT/DevGuideExamples/Multithreading/GracefulShutdown directory.

17.3.3 Request Processing

A CORBA request is processed in five stages:
1. The ORB binds the request to a thread.
2. The ORB dispatches the request to a POA.
3. The POA dispatches the request to an application object (servant).
4. The servant completes the request and returns.
5. The ORB sends a reply to the client.

Strategies employed by a multithreaded ORB as it receives and dispatches requests determine the extent to which requests may be processed concurrently. Since the CORBA specification does little to address how an ORB should select a thread for dispatching requests, different ORB implementations employ different concurrency models for request dispatching. TAO employs the following concurrency models for request dispatching:

• single-threaded—all requests are processed on a single thread.
17.3 Multithreading in the Server

- thread-per-connection—a separate thread is spawned to process all requests received via a distinct network connection. At least one thread must be available to handle incoming client connections.

- thread-pool—request processing is distributed among a group (pool) of threads.

Each concurrency model has its strengths and weaknesses, which motivates understanding an application’s request processing characteristics to choose the most appropriate strategy or combination of strategies.

The remainder of this section describes the POA’s threading models. See Chapter 11 of *Advanced CORBA Programming with C++* for more information on the Portable Object Adapter. See 17.3.3.2 for a complete discussion of TAO’s threading models.

17.3.3.1 The POA’s Threading Models

The POA threading models establish concurrency constraints that are imposed during request processing in a multithreaded environment. These constraints further qualify the extent to which CORBA requests may be processed concurrently.

A POA’s threading model is determined by its ThreadPolicy value. The CORBA specification defines three standard ThreadPolicy values:

- **ORB_CTRL_MODEL**—requests are dispatched to application objects from the thread with which the ORB invoked the POA. Concurrent servant upcalls may occur if the ORB’s strategies allow requests to be dispatched concurrently. **ORB_CTRL_MODEL** is the default ThreadPolicy setting.

- **SINGLE_THREAD_MODEL**—the POA dispatches requests to application objects sequentially. Concurrent servant upcalls cannot occur within the scope of one **SINGLE_THREAD_MODEL** POA.

- **MAIN_THREAD_MODEL**—requests for all main-threaded POAs are dispatched sequentially. Concurrent servant upcalls cannot occur within the scope of all **MAIN_THREAD_MODEL** POAs.

**Note** *TAO does not support the MAIN_THREAD_MODEL POA ThreadPolicy value.*
ORB-Controlled Model
An ORB-controlled POA places no constraints on concurrent requests; the POA effectively abdicates responsibility for threading and concurrent request processing to the ORB. Figure 17-1 shows servants activated in POAs using the ORB-controlled threading model.

![Figure 17-1 ORB-Controlled Threading Model](image)

In the ORB-controlled threading model, concurrent servant upcalls may occur at the POA level in multithreaded environments if the ORB dispatches requests concurrently. Application objects activated in an ORB-controlled POA in a multithreaded environment should be multithread safe. Ensuring that those application objects are multithread safe is the responsibility of the application developer.

Single Thread Model
A single-threaded POA constrains request processing such that concurrent upcalls cannot occur at the POA level, even in multithreaded environments. Figure 17-2 shows servants activated in POAs using the single-thread model.

A multithreaded ORB may dispatch concurrent requests to a single-threaded POA, but the subsequent servant upcalls will occur sequentially; thus an application object activated in one single-threaded POA will not be subject to concurrent requests. However, an application object activated in multiple single-threaded POAs may be subject to multiple concurrent requests in a...
multithreaded environment, as shown in Figure 17-3. Therefore, activating an application object in multiple POAs is not recommended.

Figure 17-2 Single-Thread Threading Model

Figure 17-3 Servant Activated in Multiple Single-threaded POAs
Main Thread Model
In this model, requests are dispatched to all main-thread POAs sequentially. This model effectively serializes requests dispatched by main-thread POAs at the ORB level. Figure 17-4 shows servants activated in POAs using the main-thread model.

Figure 17-4 Main-Thread Model
An application object activated by a main-thread POA will not be subject to concurrent requests. The main-thread model is also applicable in processing environments where some code must execute on the main thread. In such environments, the main-thread model insures that servant upcalls are processed on that thread. However, the application must make the main thread available to the ORB by calling either CORBA::ORB::run() or CORBA::ORB::perform_work().

Note  
* TAO does not support the MAIN_THREAD_MODEL POA ThreadPolicy value.

Configuring a POA’s Threading Model
A POA’s thread policy, like all other POA policies, can be assigned only when the POA is created. In most cases, the default ThreadPolicy of ORB_CTRL_MODEL is desired, so no intervention is required on the part of the developer. If a single-threaded environment is desired, simply give the ORB exactly one thread.
However, if you’d like to create a POA using the SINGLE_THREAD_MODEL, the remainder of this section demonstrates how to do that.

The relevant portions of the PortableServer module and the POA’s interface are:

```cpp
module PortableServer {

    // Portions of the PortableServer module omitted.

    const CORBA::PolicyType THREAD_POLICY_ID = 16;

    enum ThreadPolicyValue {
        ORB_CTRL_MODEL,
        SINGLE_THREAD_MODEL,
        MAIN_THREAD_MODEL
    };

    local interface ThreadPolicy : CORBA::Policy {
        readonly attribute ThreadPolicyValue value;
    };

    local interface POA {

        // Portions of interface POA omitted.

        POA create_POA(
            in string adapter_name;
            in POAManager a_POAManager,
            in CORBA::PolicyList policies
        ) raises (AdapterAlreadyExists, InvalidPolicy);

        ThreadPolicy create_thread_policy(
            in ThreadPolicyValue value);
    };

    The following code fragment demonstrates how to create a single-threaded POA:

    // Portions of int main(int, char *argv[]) omitted.

    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get a reference to the RootPOA.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
```
Multithreading with TAO

PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());

// Create and populate a policy list.
CORBA::PolicyList policies(1);
policies[0] = poa->create_thread_policy(PortableServer::SINGLE_THREAD_MODEL);

// Use the RootPOA’s POAManager.
PortableServer::POAManager_var mgr = poa->the_POAManager ();

// Create a child POA.
// Threading model is single-threaded.
// All other policies assume their default values.
PortableServer::POA_var st_poa = poa->create_POA("ST POA", mgr.in(), policies);

// Release memory allocated to the policy list.
policies[0]->destroy();

Guidelines for Application Developers

The ORB-controlled threading model (ORB_CTRL_MODEL) is the default, and usually the desired, threading model. It allows application developers to make the most of an ORB’s concurrent processing capabilities and performance optimization strategies. An application using this threading model can also take advantage of fine-grained contention management techniques to resolve application objects’ contention for shared resources and services. However, this model obligates application developers to assume concurrent requests will occur, and to identify and resolve points of contention that may arise during concurrent CORBA requests.

The following scenarios might motivate use of the ORB-controlled threading model for a CORBA-based application that is subject to concurrent requests:

- The application’s mission is to provide access to and resolve contention for a shared resource or service, i.e. the application is effectively a contention manager so using an ORB-controlled threading model does not add additional complexity.
- The application serializes and dispatches events, perhaps received from disparate sources, to discrete event processors (application objects) so multiple events can be processed concurrently. As with the previous scenario, applications such as this are obligated to resolve contention.
- The application is a CORBA interface to an existing system that permits concurrent invocations.
• Acceptable performance can only be achieved with concurrent processing and fine-grained contention management.

• The application is stateless, i.e. the context carried with each CORBA request is sufficient to process the request.

The single-threaded model (SINGLE_THREAD_MODEL) can simplify an application’s design and implementation by guaranteeing that an application object will not be subject to concurrent invocations from a single POA. Applied within its constraints, this model shifts some responsibility for contention management away from the application.

However, this mechanism can only resolve contention for distinct servants arising from concurrent requests; it cannot resolve contention for services and resources shared by application objects. Resolving contention for commonly-used services and resources remains the application’s responsibility.

The single-threaded model is appropriate when:

• The likelihood of concurrent requests is acceptably low.

• The increased latency caused by the serialization of requests is acceptable.

• The potential performance improvement resulting from concurrent request processing does not justify the added complexity associated with fine-grained contention management.

• An application depends upon software libraries or legacy code that effectively prohibits concurrent processing.

The main-thread model (MAIN_THREAD_MODEL) is applicable in two circumstances:

• An application’s environment mandates that certain portions of code execute only on the main thread. However, the application is obligated to make the main thread available to the ORB by calling run() or perform_work() on the ORB from the main thread.

• An application requires protection from concurrent requests more rigorous than that provided by the single-threaded model.

This model imposes the most stringent constraints upon an application and likely reduces a multithreaded ORB’s efficiency. Application developers may want to compare performance of this threading model with that of a single-threaded ORB before committing to this model.
**Multithreading with TAO**

**Note**  
*TAO does not support the MAIN_THREAD_MODEL POA ThreadPolicy value.*

The following section describes usage of the default ORB_CTRL_MODEL thread policy with TAO’s threading models.

### 17.3.3.2 TAO’s Threading Models

The ORB-controlled threading model (ORB_CTRL_MODEL) allows application developers to make the most of an ORB’s concurrent processing capabilities and performance optimization strategies by enabling the application developer to take advantage of TAO’s server-side concurrency models. TAO’s concurrency models are relevant only to a POA configured with the default ORB_CTRL_MODEL thread policy.

In this section, we discuss TAO’s various concurrency models. Concurrency in TAO is controlled by various factors, including how the ORB is configured at run time, how many threads are in the process, and the specific behavior of each thread.

TAO provides three concurrency models affecting how a server thread receives and processes requests:

- reactive
- thread-per-connection
- thread-pool

**Note**  
*In addition to these, TAO’s support of Real-Time CORBA provides the ability to associate a thread pool with a POA; see 9.3.7 for more information on using RT CORBA’s thread-pool policy.*

The remainder of this section discusses each of these concurrency models in detail. For each concurrency model, we present the motivation for its use, configuration options to enable it, and consequences of its use. We have provided a working example for each concurrency model.

**Reactive Concurrency Model**

By default, TAO servers process incoming client connections and requests *reactively*. Typically, a single thread (often the main thread) is dedicated to
processing client requests by calling `CORBA::ORB::run()`. Other threads may also exist in the process, but are dedicated to performing other tasks.

In the reactive concurrency model, the ORB uses an ACE Reactor to receive and dispatch client requests. The default reactor type used in TAO is the thread-pool reactor, but the ORB can be configured at run time to use a different reactor type (see 17.3.3.3 and 20.7.6). Assuming that the ORB’s `run()` operation is called from only one thread, that single thread will be used for receiving and dispatching all requests.

**Note**  
*If a client thread makes an outbound request, that thread may be used to process inbound requests while waiting for its reply. When this occurs, the thread is said to be in a “nested upcall.”* See 17.4.4 and 22.2.3 for more details.

**Note**  
*More information on the ACE Reactor Framework and ACE Reactor implementations can be found in Chapters 3 and 4 of *C++ Network Programming: Volume 2 (C++NPv2)* and Chapter 7 of *The ACE Programmer’s Guide (APG).*

Figure 17-5 illustrates a server using the reactive concurrency model with a single thread. All incoming requests are received and processed on the same thread. Subsequent requests, either from the same client or a different client,
are blocked waiting for the thread to complete processing the previous request.

Motivation
The reactive concurrency model is useful in the following situations:

- The application is a single-threaded server in which request processing times remain relatively fixed.
- Since this model is simple to configure and program, it is often used during initial development and testing of interface implementations.

Configuration
The reactive concurrency model is selected by default in TAO; no specific action is necessary to configure it. However, it can be specified explicitly by supplying the following configuration option to the default server strategy factory:

```
static Server_Strategy_Factory "-ORBConcurrency reactive"
```

See 21.3.5 for more information on using this option.

Consequences
The consequences of using the single-threaded reactive concurrency model are as follows:
17.3 Multithreading in the Server

- Requests are processed in the order received.
- Subsequent requests may be blocked until the thread completes processing the previous request. This can lead to unbounded latencies.
- Since requests are not processed concurrently, developers may be able to simplify synchronization control in their implementation code. However, synchronization issues should not be completely ignored in case multithreading is used elsewhere in the application or added later.
- Certain locks in the ORB and reactor may be disabled to improve throughput.
- Scalability is limited; as the number of simultaneous client requests increases, so does the per-request latency.

Example
The following example shows a typical single-threaded server using the reactive concurrency model. Configuration of the reactive concurrency model has been specified explicitly even though it is the default model. In this example, we have also configured the single-threaded reactor type (-ORBReactorType select_st), the single-threaded wait strategy (-ORBWaitStrategy st), and disabled some locks to attempt to maximize overall throughput. See 17.3.3.3 for more information on the single-threaded reactor. See 17.4.4.2 for more information on the single-threaded, wait-on-reactor wait strategy.

The server source code for this example is the Messenger server from Chapter 3. We show only the file MessengerServer.cpp, which contains the code for the server’s main() function. Only one thread exists in the server and that thread is dedicated to processing client connections and requests by calling CORBA::ORB::run(). The source code for this example is in the $TAO_ROOT/DevGuideExamples/Multithreading/Reactive directory.

```cpp
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```
Multithreading with TAO

// Get a reference to the RootPOA.
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

// Activate the POAManager.
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Create a servant.
Messenger_i messenger_servant;

// Register the servant with the RootPOA, obtain its object reference, stringify it, and write it to a file.
PortableServer::ObjectId_var oid =
    poa->activate_object(&messenger_servant);
CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
CORBA::String_var str = orb->object_to_string(messenger_obj.in());
std::ofstream iorFile("Messenger.ior");
iorFile << str << std::endl;
iorFile.close();
std::cout << "IOR written to file Messenger.ior" << std::endl;

// Accept requests from clients.
orb->run();
orb->destroy();
}
catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
    return 1;
}

return 0;
}

The service configurator file, shown below, contains directives to configure the reactive concurrency model, use the single-threaded select reactor type, disable certain locks, and use the single-threaded wait-on-reactor wait strategy. These options are selected to improve overall throughput. Each directive should appear on a single line.

# server.conf file for single-threaded reactive server.
static Server_Strategy_Factory "-ORBConcurrency reactive -ORBPOALock null"
dynamic Advanced_Resource_Factory Service_Object *
TAO_Strategies: make_TAO_Advanced_Resource_Factory () "+ORBReactorType select_st
-ORBInputCDRAllocator null"
static Client_Strategy_Factory "+ORBProfileLock null -ORBWaitStrategy st"
In the above example, we use the dynamic directive to configure the Advanced Resource Factory so it can be loaded and configured from the TAO_Strategies library dynamically. Alternatively, we can statically link the application with the TAO_Strategies library, insert the line:

```
#include <tao/Strategies/advanced_resource.h>
```

prior to `main()`, and use the static directive to configure the Advanced Resource Factory:

```
static Advanced_Resource_Factory "-ORBReactorType select_st -ORBInputCDRAllocator null"
```

See 20.5 for more information on using and configuring TAO’s Advanced Resource Factory.

**Thread-per-Connection Concurrency Model**

The simplest multithreaded server configuration in TAO is the thread-per-connection concurrency model. In this model, the ORB automatically spawns a new thread for each new client connection. Each spawned thread is dedicated to processing requests that arrive on a given connection. The thread reads and processes those requests serially. When the connection is closed, the thread associated with that connection exits and all resources allocated by the ORB associated with that thread are released. At least one thread (other than those that are dedicated to processing requests) must remain available to the ORB to handle new connections; usually, this is accomplished by calling `CORBA::ORB::run()` in the main thread.
Figure 17-6 shows a multithreaded server using the thread-per-connection concurrency model.

**Figure 17-6 Thread-per-connection concurrency model**

**Motivation**
The thread-per-connection concurrency model is useful in the following situations:

- Multiple client connections are active simultaneously.
- Request processing is computationally intense.
- Requests arriving over separate connections are largely decoupled from one another.

**Configuration**
The thread-per-connection concurrency model is specified by supplying the following configuration option to the default server strategy factory:

```
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection"
```

See 21.3.5 for more information on using this option.

**Consequences**
The consequences of using the thread-per-connection concurrency model are as follows:
17.3 Multithreading in the Server

- The total number of threads created and managed by the ORB is one plus the number of simultaneous client connections.
- Requests arriving on different connections are processed concurrently.
- Requests arriving on a given connection are processed serially, always by the same thread.
- Requests from one client are not blocked by requests from another client.
- Subsequent requests from one client connection may be blocked until the thread dedicated to that connection completes processing the previous request.
- Since requests arriving on different connections may be processed concurrently, developers may have to protect critical regions of code with locks. However, depending upon the application, it may be possible to reduce locking overhead. For example, if processing of requests on different connections is completely decoupled, with no shared code or shared state, locking may not be necessary.
- Threads that are spawned by the ORB for each new connection do not use a reactor for receiving and dispatching inbound requests; they simply block waiting to read. These threads may need to awaken periodically to check whether the ORB has been shut down. The default server strategy factory’s `-ORBThreadPerConnectionTimeout` option can be used to configure the thread wake-up interval. See 21.3.11 for more information on this option.
- All threads spawned by the ORB process requests at the same priority. Applications that need to process different requests at different priorities should use the features provided by the RT CORBA thread-pool model. See 9.3.7 for more information.
- Scalability is limited; as the number of simultaneous incoming connections increases, so does the number of threads in the server.

Note: A single client may open multiple simultaneous connections to a server ORB, depending upon the number of client threads that invoke requests and the configuration of the ORB’s transport multiplexing strategy (`-ORBTransportMuxStrategy`). Thus, using the thread-per-connection concurrency model, it is possible for multiple server threads to be created for
processing requests from a single client. See 17.4.2 and 22.3.3 for more information on the -ORBTransportMuxStrategy option.

Example

In this section, we describe an example of a typical multithreaded server using the thread-per-connection concurrency model. No application code changes are required for creating and managing threads; all thread management is performed by the ORB. However, the application’s servant implementations may need to provide locking around critical regions of code.

The server source code for the example is the Messenger server from Chapter 3. The main thread is dedicated to processing client connections by calling CORBA::ORB::run(). The ORB automatically spawns a new thread for each new connection. Since no changes are required to the server’s main() function, we do not show the file MessengerServer.cpp.

The source code for this example is in the $TAO_ROOT/DevGuideExamples/Multithreading/ThreadPerConnection directory.

The server’s service configurator file, shown below, contains directives to configure the ORB to use the thread-per-connection concurrency model. It also shows how the thread-per-connection time-out interval (in milliseconds) can be optionally configured at run time. In this example, we wake up the server’s threads on a one-second interval so they can check if the ORB has been shut down. Each directive should appear on a single line.

```
# server.conf file for thread-per-connection server.
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection
-ORBThreadPerConnectionTimeout 1000"
```

Thread-pool Concurrency Model

In the thread-pool concurrency model, a pool of threads is made available to the ORB for handling incoming client connections and processing inbound requests. The developer takes responsibility for creating the threads and calling CORBA::ORB::run() or CORBA::ORB::perform_work() on the same ORB instance from each thread. Connections are established and requests are dispatched onto threads in the pool by the thread-pool reactor, which is described in 17.3.3.3. The thread-pool reactor allows multiple threads to handle events concurrently. If a thread is available to handle an event, then
the event is dispatched right away; otherwise, it is held until a thread becomes ready.

The thread-pool reactor implements the *Leader/Followers* architectural pattern described in *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (POSA2), by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. The leader-followers model allows multiple threads to share a single reactor. One thread becomes the “leader” and runs the reactor’s event loop. The leader thread receives and processes the next event, such as an incoming client connection or inbound request. Other threads in the pool are “followers.” As soon as the leader thread receives an event, one of the follower threads becomes the new leader. When a thread finishes processing an event, it rejoins the leader-followers group and waits to become the leader thread again.

Since TAO uses the thread-pool reactor by default, any TAO server can be easily configured to use a thread pool by simply spawning a number of threads and calling `CORBA::ORB::run()` on the same ORB instance in each thread.

Figure 17-7 shows a server using the thread-pool concurrency model. In this server, four threads have been spawned by the application and made available to the ORB for request processing.
**Motivation**
The thread-pool concurrency model is useful in the following situations:

- Multiple client connections can be active simultaneously.
- Multiple threads can be dedicated to processing requests.
- The number of active client connections can be large, thereby making the thread-per-connection model undesirable due to the large number of threads that would be spawned by the ORB.
- Multiple concurrent requests may be received on a single connection (e.g., from a multithreaded client that is multiplexing requests on a single connection) and need to be processed concurrently.
- Request processing is computationally intense.

**Configuration**
Since TAO uses the thread-pool reactor and the reactive concurrency model by default, no special configuration options are required.

**Consequences**
The consequences of using the thread-pool concurrency model are as follows:

- The total number of threads used for processing requests is bounded and can be controlled by the server developer.
- The server developer must take responsibility for creating threads and running the ORB’s event loop in each thread.
- Requests are processed concurrently, regardless of the connection on which they arrive.
- Requests from one client will not be blocked by requests from another client.
- Subsequent requests from one client connection will not necessarily be blocked waiting for processing of a previous request on that connection to complete.
- Since requests may be processed concurrently, developers may have to protect critical regions of code with locks.
- Since there is no way to predict which thread in the pool will be used to process a given request, all threads in the pool should be run at the same priority. Applications that need to process different requests at different
priorities should use the features provided by the RT CORBA thread-pool model. See 9.3.7 for more information.

- Scalability is improved, especially for servers with large numbers of clients, not all of which are sending requests simultaneously.

**Example**
The following example shows a typical multithreaded server using the thread-pool concurrency model. Since the application is responsible for creating threads, application code changes are required to use the thread-pool model. In addition, the application's servant implementations may need to provide locking around critical regions of code.

The server source code for this example is based on the Messenger server from Chapter 3. The main thread creates multiple threads and dedicates each to processing requests by calling CORBA::ORB::run(). The ORB dispatches requests onto these threads using the ACE thread-pool reactor. The server's main() function is implemented in the file MessengerServer.cpp. The source code for this example is in the $TAO_ROOT/DevGuideExamples/Multithreading/ThreadPool directory.

```cpp
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

// 1. Define a "task" class for implementing the thread-pool threads.
#include <ace/Task.h>

class ORB_Task : public ACE_Task_Base
{
public:
    ORB_Task (CORBA::ORB_ptr orb)
    : orb_(CORBA::ORB::_duplicate(orb)) { }
    virtual ~ORB_Task () { }
    virtual int svc ()
    {
        this->orb_->run();
        return 0;
    }
private:
    CORBA::ORB_var orb_; 
};

// 2. Establish the number of threads.
```
static const int nthreads = 4;

int main(int argc, char* argv[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Get a reference to the RootPOA.
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

        // Activate the POAManager.
        PortableServer::POAManager_var mgr = poa->the_POAManager();
        mgr->activate();

        // Create a servant.
        Messenger_i messenger_servant;

        // Register the servant with the RootPOA, obtain its object
        // reference, stringify it, and write it to a file.
        PortableServer::Object_id_var oid =
            poa->activate_object(&messenger_servant);
        CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
        CORBA::String_var str = orb->object_to_string(messenger_obj.in());
        std::ofstream iorFile("Messenger.ior");
        iorFile << str << std::endl;
        iorFile.close();
        std::cout << "IOR written to file Messenger.ior" << std::endl;

        // 3. Create and activate threads for the thread pool.
        ORB_Task task (orb.in());
        int retval = task.activate (THR_NEW_LWP | THR_JOINABLE, nthreads);
        if (retval != 0) {
            std::cerr << "Failed to activate " << nthreads << " threads." << std::endl;
            return 1;
        }

        // 4. Wait for threads to finish.
        task.wait();

        // Clean up.
        orb->destroy();
    } catch (CORBA::Exception& ex) {
        std::cerr << "CORBA exception: " << ex << std::endl;
        return 1;
    }

    return 0;
The code sections numbered 1-4 in the above example have been added to provide a thread pool for request processing by the ORB. Each step is further described here:

1. **Define a “task” class for implementing the thread-pool threads.**
   Threads for the thread pool can be created using the platform’s normal threading APIs (e.g., `pthread_create()` on platforms that support POSIX threads). However, ACE provides an object-oriented abstraction for threads known as *task*. In this section of the code, we define a new task type, called `ORB_Task`, that is dedicated to running the ORB’s event loop. We can initialize an `ORB_Task` with an object reference to the ORB, then activate several threads. Each thread enters the task’s `svc()` method, in which we simply call `run()` on the ORB to make that thread available to the ORB for handling connections and requests. If the ORB is shut down, then all the threads exit the ORB’s event loop, causing each `svc()` method to return and each thread to exit.

2. **Establish the number of threads.**
   We define a constant to represent the number of threads we want in the thread pool. With a few simple code changes, this value could be provided as a command-line parameter to allow the number of threads in the thread pool to be dynamically configured at run time.

3. **Create and activate threads for the thread pool.**
   In this section of the code, we create an instance of our `ORB_Task` class, then create a number of threads using the task’s `activate()` method. Each new thread invokes the task’s `svc()` method. The thread creation flags are a bitwise OR of the values `THR_NEW_LWP` and `THR_JOINABLE`. The `THR_NEW_LWP` flag means a new light-weight process (LWP) is created for each new thread to increase the concurrency level of the thread pool. `THR_JOINABLE` means we can later join these threads and wait for each one to exit using the task’s `wait()` method.

4. **Wait for threads to finish.**
   Since all ORB request processing is taking place in the threads activated via the `ORB_Task`, the main thread continues. To keep it from exiting (and possibly causing the process to exit), we wait for all the threads in the pool to finish.
to exit. After the last thread in the thread pool has exited, we can proceed to destroy the ORB and clean up resources.

**Note**  
More information on the ACE Task Framework can be found in Chapter 6 of *C++ Network Programming: Volume 2 (C++NPv2)* and Chapter 15 of *The ACE Programmer’s Guide (APG)*.

### 17.3.3.3 Configuring TAO’s Reactor

Under the covers, TAO uses a *reactor* to handle events from clients. A reactor is an object-oriented layer over an operating system’s event handling functions. The reactor separates the detection of events from the handling of those events. Applications may register event handlers with the reactor whereby they are associated with various handles or other sources of events (e.g., timers, signals). When an event occurs on one of these handles or other event sources, the reactor dispatches the event to the associated event handler. The ORB uses a reactor to accept new connections in a socket context and to respond to incoming and outgoing data.

**Note**  
For more information on the reactor, see Chapters 3 and 4 of *C++ Network Programming: Volume 2 (C++NPv2)* and Chapter 7 of *The ACE Programmer’s Guide (APG)*.

There are several different reactor implementation types available to the ORB. The Advanced Resource Factory’s `-ORBReactorType` option allows run-time configuration of the type of reactor that the ORB uses to handle events. See 20.7.6 for more information on using this option. Here we discuss how the selection of reactor type affects the ORB’s threading behavior. We discuss only the single-threaded select reactor, multithreaded select reactor, and thread-pool reactor implementations.

**Single-threaded Select Reactor**

As described in 17.3.3.2, single-threaded applications that are dedicated to receiving and processing CORBA requests can benefit from using the single-threaded select reactor type (`-ORBReactorType select_st`). Since all request processing is performed on a single thread, these applications may also safely disable certain locks. The single-threaded select reactor uses the
select() system call to monitor I/O handles for input and output availability. It dispatches all inbound client connections and requests onto a single thread. If the thread is busy, new connections and requests are blocked in the transport layer until the thread re-enters the reactor’s event loop.

The thread re-enters the reactor’s event loop when it has completed processing each connection or request. If the application makes an outbound (client role) request within the context of processing an inbound (server role) request, then it may enter the reactor’s event loop to await its reply (depending upon the wait strategy); while waiting for its reply in this fashion, the thread may be used by the reactor to dispatch additional inbound requests or other events. See 17.4.4 for more information on client-side wait strategies.

Use the Advanced Resource Factory’s -ORBReactorType select_st option to configure the single-threaded select reactor type, as shown here:

dynamic Advanced_Resource_Factory Service_Object *
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBReactorType select_st"

The above directive should appear on one line in the application’s service configurator file.

### Multithreaded Select Reactor

Like the single-threaded select reactor, the multithreaded select reactor uses the select() system call to monitor I/O handles for input and output availability. However, this reactor type also employs locks to enforce synchronization between threads. Only one thread is allowed to enter the reactor’s event loop at a time. When an event is dispatched onto a thread, that thread retains ownership of the reactor (i.e., holds the reactor’s token) and another thread is not allowed to enter into the reactor’s event loop until the owner thread finishes dispatching all active event handlers. While a thread holds the reactor token, other threads cannot get into select(). A single thread may recursively call the reactor’s handle_events() method without deadlocking.

In summary, the select_mt reactor type, while safe to use in multithreaded servers, performs poorly compared to the thread-pool reactor due to its coarse level of synchronization. Use of the select_mt reactor will likely result in more jittery performance compared to the thread-pool reactor. Multithreaded applications should use the thread-pool reactor, which is discussed below.
Thread-Pool Reactor
The default reactor type in TAO is the thread-pool (tp) reactor. The underlying implementation of the thread-pool reactor uses the ACE_TP_Reactor type, a specialization of the ACE_Select_Reactor that is designed to support thread-pool-based event dispatching. Since the select reactor receives events via the select() system call, only one thread can be blocked in the reactor’s handle_events() method at a time. The thread-pool reactor overcomes this limitation by taking advantage of the fact that events reported by select() are held (e.g., in transport-level input buffers) if not acted upon immediately. The thread-pool reactor keeps track of which event handler was most recently activated, releases the reactor’s internal lock (thereby allowing another thread to enter the event loop), then dispatches the event to the event handler outside the context of the lock.

To simplify synchronization within event handlers, the thread-pool reactor ensures that a given event handler cannot be called by multiple threads simultaneously. It does this by automatically suspending an activated event handler before making the upcall onto its event-handling code (e.g., the handle_input() method), then resuming the event handler after the upcall completes.

Note
The ACE_TP_Reactor implements the Leader/Followers architectural pattern described in Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2). More information on the ACE_TP_Reactor can also be found in Chapter 4 of C++ Network Programming: Volume 2 (C++NPv2).

Synchronization among event processing threads in the thread-pool reactor is far more efficient than in the multithreaded select (select_mt) reactor. Any TAO server that uses the thread-pool concurrency model, as described in 17.3.3.2, should use the thread-pool reactor. Since the thread-pool reactor is the default reactor type, no explicit configuration options are necessary. However, the thread-pool reactor type can be specified explicitly via the Advanced Resource Factory’s -ORBReactorType tp option, as shown here:

dynamic Advanced_Resource_Factory Service_Object *
TAO_Strategies: make_TAO_Advanced_Resource_Factory () "-ORBReactorType tp"
17.4 Multithreading in the Client

The above directive should appear on one line in the application’s service configurator file.

17.3.3.4 Flushing Replies to the Client
When a server thread sends a reply to a client, the reply message is marshaled for writing to the transport. The ORB uses a resource factory to obtain a strategy for flushing these messages. In some cases, messages prepared for writing may be queued and written to the transport at a later time. For more information on flushing strategies, please see the discussion in 17.4.3.

17.4 Multithreading in the Client

The client-side ORB has several configuration options that affect the client’s threading behavior. In this section, we describe the effects of various configuration options under various conditions and provide guidelines on making appropriate configuration choices to meet your application’s needs.

As described in 17.2, a thread may behave as a client (invokes requests, waits for replies) or a server (processes requests, sends replies). Under certain conditions, a thread may switch between the client and server roles. The discussion of concurrency in this section pertains to a thread in the role of a client.

In the remainder of this section, we describe options for controlling the following ORB strategies:

- **Establishing a connection to the server**
  When a client thread invokes a request on a remote CORBA object, the client-side ORB must have a transport-level connection to the target object’s ORB over which to send the request message. TAO’s `connect` strategy determines how the client thread waits for the connection to be completed.

- **Multiplexing requests on a connection**
  When a client thread invokes a request on a remote CORBA object, the ORB may be able to reuse an existing connection or it may have to establish a new connection. It is possible configure TAO’s `transport multiplexing strategy` to multiplex several simultaneous requests on a single connection.
• **Flushing requests to the server**

When a client thread sends a request or a server thread sends a reply, the request or reply message is first marshaled for writing to the transport. Messages prepared for writing may be queued and written to the transport at a later time as determined by TAO’s *flushing strategy*.

• **Waiting for a reply from the server**

When a client thread invokes a synchronous request on a remote CORBA object, it must wait for the reply. The way in which the client thread waits for its reply and the activities the thread may perform in the meantime is determined by TAO’s *wait strategy*.

• **Optimizing performance of collocated requests**

When a client thread invokes an operation on a collocated CORBA object (i.e., an object that is implemented within the same address space as the client), the invocation may be optimized to bypass several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic by configuring TAO’s *collocation optimization*.

### 17.4.1 Establishing a Connection to the Server

When a client thread invokes a request on a remote CORBA object, the client-side ORB must have a transport-level connection to the target object’s ORB over which to send the request message. The ORB may be able to reuse an existing connection, or it may have to establish a new connection. Establishing a new connection can be time-consuming. TAO’s default client strategy factory provides a *connect strategy* that determines how the client thread waits for the connection to be completed.

The default client strategy factory provides three possible connect strategies:

• **leader-follower (lf)**

In the leader-follower connect strategy, the connecting thread joins the leader-follower group of threads to be awakened when the connection has been completed.

While waiting for the connection to complete, the thread may become the leader and may be used to process inbound requests or other events. Thus, with the leader-follower connect strategy, it is possible for a client thread to temporarily switch to the server role.
TAO uses the leader-follower connect strategy by default.

- reactive

The reactive connect strategy behaves similarly to the leader-follower connect strategy, except that the connecting thread uses the ORB’s reactor instead of joining the leader-follower group of threads. It provides non-blocking connect behavior with less locking overhead than the leader-follower connect strategy.

The thread registers an event handler with the reactor, then runs the reactor’s event loop. The event handler is invoked by the reactor when the connection is complete. The reactive connect strategy should only be used in single-threaded or pure client applications.

- blocking (blocked)

In the blocking connect strategy, the connecting thread simply blocks for the duration of the connection attempt. This option should only be used in multithreaded environments or in environments where network congestion is unlikely to be a factor.

The connect strategy can be configured at run time by setting the default client strategy factory’s -ORBConnectStrategy option in the application’s service configurator file. This option accepts one of three values: lf (leader-follower), reactive, or blocked. The default setting for this option is lf.

In the example below, we show the directive to select the blocked strategy.

```
static Client_Strategy_Factory "-ORBConnectStrategy blocked"
```

See 22.3.1 for more information on using the -ORBConnectStrategy option.

---

**Note**

The blocked connect strategy is required when you use the wait-on-read wait strategy (-ORBWaitStrategy rw). See 17.4.4.1 for more information on wait-on-read.

## 17.4.2 Multiplexing Requests on a Connection

When a client thread invokes a request on a remote CORBA object, the client-side ORB must have a transport-level connection to the target object’s
ORB over which to send the request message. The ORB may be able to reuse an existing connection, or it may have to establish a new connection.

If a reply is expected, the request is said to be pending on the connection. Under some conditions, only one request may be pending on a connection at a time. For example, if the client is using the wait-on-read wait strategy as described in 17.4.4.1, there cannot be more than one pending request on a single connection. In other cases, it is possible to multiplex requests on a single connection. Multiplexing requests on a single connection can optimize utilization of connection-related resources between a client and a server.

The default client strategy factory provides two possible transport multiplexing strategies:

- **exclusive**

  An exclusive transport uses a connection to service a single request and receive a reply before making it available for another request. A multithreaded client using the exclusive transport strategy will open a new connection for each concurrent request.

- **multiplexed (muxed)**

  The multiplexed strategy allows multiple concurrent requests to share a single connection, using asynchronous callbacks to handle the distribution of the replies.

  TAO uses the multiplexed transport multiplexing strategy by default.

The transport multiplexing strategy can be configured at run time by setting the default client strategy factory’s `-ORBTransportMuxStrategy` option in the application’s service configurator file. This option accepts one of two values: `muxed` (requests can be multiplexed on a connection) or `exclusive` (each request requires exclusive access to a connection). The default setting for this option is `muxed`.

In the example below, we show the directive to select the exclusive transport multiplexing strategy.

```cpp
static Client_Strategy.Factory "-ORBTransportMuxStrategy exclusive"
```

See 22.2.2 for more information on using the `-ORBTransportMuxStrategy` option.
### 17.4.3 Flushing Requests to the Server

When a client thread sends a request or a server thread sends a reply, the request or reply message is first marshaled for writing to the transport. The ORB then uses a resource factory to obtain a strategy for flushing these messages. In some cases, messages prepared for writing may be queued and written to the transport at a later time, according to the **flushing strategy**. The default resource factory provides three possible message flushing strategies:

- **leader-follower** *(leader_follower)*

  In the leader-follower flushing strategy, the thread sending the message will first attempt a non-blocking write operation to the transport. If the write operation succeeds, the thread will continue. If the non-blocking write operation could not be completed, the message is queued and the thread joins the leader-follower group of threads to be awakened when it can flush its message to the transport.

  While waiting to flush queued messages, the thread may become the leader and may be used to process inbound requests or other events. Thus, with the leader-follower flushing strategy, it is possible for a client thread to temporarily switch to the server role, or for a server thread flushing a reply message to be used to process additional requests.

  TAO uses the leader-follower flushing strategy by default.

- **reactive**

  The reactive flushing strategy behaves similarly to the leader-follower flushing strategy, except that a sending thread uses the ORB’s reactor instead of joining the leader-follower group of threads. It provides non-blocking flushing with less locking overhead than the leader-follower flushing strategy.

  First, a non-blocking write operation to the transport is attempted. If it is not successful, then the thread queues the message, registers an event handler with the reactor, and runs the reactor’s event loop. When writing to the transport is possible, the thread flushes its queued message.
The reactive flushing strategy should only be used in single-threaded or pure client applications.

- **blocking**

In the blocking flushing strategy, a thread sending a message simply blocks until it has successfully written its message to the transport. While blocked waiting to write, the thread will not be available to the ORB for connection handling or request processing. The blocking flushing strategy should only be used in pure client applications or in environments where network congestion is unlikely to be a factor.

The flushing strategy can be configured at run time by setting the default resource factory’s `-ORBFlushingStrategy` option in the application’s service configurator file. This option accepts one of three values: `leader_follower`, `reactive`, or `blocking`. The default setting for this option is `leader_follower`.

In the example below, we show the directive to select the `blocking` strategy.

```plaintext
static Resource_Factory "-ORBFlushingStrategy blocking"
```

See 20.6.6 for more information on using the `-ORBFlushingStrategy` option.

---

**Note**  
The **blocking flushing strategy is required when you use the wait-on-read wait strategy** (`-ORBWaitStrategy rw`). See 17.4.4.1 for more information on wait-on-read.

---

### 17.4.4 Waiting for a Reply from the Server

When a client thread invokes a synchronous request on a remote CORBA object, it must wait for a reply. The way in which the client thread waits for its reply and what it might do in the meantime is determined by the application’s **wait strategy**.

By default, TAO attempts to use all available threads efficiently. This includes client threads that have sent a request to a server and are waiting for a reply. While waiting, a client thread may be “borrowed” by the ORB to handle incoming requests, performing a **nested upcall**. If an application acts in both the client and server roles, then nested upcalls are possible.
This “borrowing” of a client thread waiting for a reply has side effects that may be surprising. First, it is possible for even a single-threaded server to process more than one request at the same time. Second, any thread known by the ORB may play the role of a client thread or a server thread.

This is TAO’s default behavior for a number of reasons. First, it avoids many potential deadlock situations. For example, suppose a single-threaded client provides a callback object to a server. After the client sends a request to the server, the server sends a callback request back to the client. If the client’s only thread is waiting for the reply and cannot handle incoming requests, then a deadlock occurs. TAO’s default behavior, on the other hand, allows the client ORB to process the callback request and send a reply to the server, freeing the server ORB to send its reply back to the client.

Second, this behavior gives the application complete control over thread creation and destruction. Some ORB implementations spawn a new thread to handle each incoming request. TAO does not do this.

A nested upcall happens in the context of an ORB. In a process with more than one ORB, a client thread that makes a request through an ORB is not asked to handle a nested upcalls for an object in another ORB.

This behavior is configurable. TAO provides several wait strategies affecting what a client thread might do while waiting for a reply. They are as follows:

- wait-on-read
- wait-on-reactor
- wait-on-leader-follower
- wait-on-leader-follower-no-upcall (experimental)

TAO uses the wait-on-leader-follower wait strategy by default.

### 17.4.4.1 Wait-on-Read Wait Strategy

In the *wait-on-read* wait strategy (also known as “receive-wait”), when a client thread invokes a synchronous request, it simply blocks waiting to read the reply message from the underlying transport. When the reply message arrives, the thread becomes unblocked and processing continues.
Figure 17-8 shows a client using the wait-on-read wait strategy.

![Diagram of client using the wait-on-read wait strategy]

**Figure 17-8 Client using the wait-on-read wait strategy**

While a client thread is blocked waiting for its reply, it is not available to the ORB for handling incoming connections or requests. In some situations, this behavior can lead to deadlocks.

For example, in a hybrid application (that plays both server and client roles), if all threads are acting in the client role and are currently blocked waiting for replies, the application is unable to respond to incoming requests from other clients. If a client thread invokes a request on a server that is also using the wait-on-read wait strategy, and the server thread that is handling that request switches to the client role and calls back to our application, threads in both
processes are now blocked waiting for their replies and neither can make progress. This situation is shown in Figure 17-9.

**Figure 17-9 Deadlock created by using wait-on-read wait strategy**

In the situation shown in Figure 17-9, the deadlock results because there are insufficient threads available to the ORB (in the process shown as “Client-Server-1”) for processing inbound requests that arrive while client threads are blocked waiting for replies. Applications can avoid this problem by making sure there is always at least one thread available to the ORB via CORBA::ORB::run() or CORBA::ORB::perform_work(). This can be accomplished using the thread-per-connection or thread-pool (with a sufficient number of threads in the pool) server concurrency models described in previous sections.

**Motivation**

The wait-on-read wait strategy is useful in the following situations:

- The application is a *pure* client application.
- All outbound request invocations are oneway requests with no replies.
- The application must ensure that client threads that are waiting on replies to outbound synchronous requests cannot be used by the ORB for processing inbound requests (i.e., nested upcalls cannot be allowed).
Multithreading with TAO

- Requests cannot be multiplexed across a single connection (i.e., the transport multiplexing strategy is configured as exclusive).
- Request messages must be completely written to the transport (i.e., the flushing strategy is configured as blocking).

Configuration
The wait-on-read wait strategy is specified by supplying the following configuration options to the default client strategy factory:

```
static Client_Strategy_Factory "-ORBWaitStrategy rw -ORBTransportMuxStrategy exclusive -ORBConnectStrategy blocked"
static Resource_Factory "-ORBFflushingStrategy blocking"
```

Each of the above directives should appear on one line in the application’s service configurator file.

See 22.3.4, 22.3.3, 22.3.1, and 20.6.6 for more information on using these options.

---

**Note** The exclusive transport multiplexing strategy (-ORBTransportMuxStrategy exclusive), the blocked connect strategy (-ORBConnectStrategy blocked), and the blocking message flushing strategy (-ORBFflushingStrategy blocking) are required when using the wait-on-read (rw) wait strategy.

---

Consequences
The consequences of using the wait-on-read wait strategy are as follows:

- Client threads that are waiting for replies cannot be “stolen” by the ORB and used to process incoming requests; nor can they be used to handle other reactor events, such as timers.
- An application that uses the wait-on-read wait strategy may not use Asynchronous Method Invocation (AMI). Since the connection handler is not registered with the ORB’s reactor, the ORB will not be able to dispatch the reply when it arrives.
- An application that uses the wait-on-read wait strategy may have a slightly smaller footprint than it would if it used the other wait strategies because there is no need for the ORB to create a reply dispatcher to handle the reply message. On the other hand, since requests cannot be
multiplexed across connections, the application could experience an increase in footprint since multiple connections may be needed when sending concurrent requests to the same target server.

### 17.4.4.2 Wait-on-Reactor Wait Strategy

In the *wait-on-reactor* wait strategy (also known as “reactive”), a client thread waits for its reply by registering its connection handler as an event handler with the ORB’s reactor, then running the reactor’s event loop (`ACE_Reactor::handle_events()`) until the reply message arrives. When the reply is received, the connection handler is invoked by the reactor, the reply is read and demarshaled, and execution continues after the point of invocation.

**Motivation**

The wait-on-reactor wait strategy is useful in the following situations:

- The application is a *pure* client application.
- The application is single-threaded and acts as both a client and a server.
- Nested upcalls are possible.

**Configuration**

The wait-on-reactor wait strategy is specified by supplying the following configuration option to the default client strategy factory:

```
static Client_Strategy_Factory "-ORBWaitStrategy st"
```

See 22.3.4 for more information on using this option.

**Consequences**

The consequences of using the wait-on-reactor wait strategy are as follows:

- A single-threaded pure client application can still handle other reactor events (e.g., timers) while waiting for replies.
- In single-threaded applications, the wait-on-reactor wait strategy still allows the application to handle inbound requests while it is waiting for replies, but the locking overhead is less than with the default wait-on-leader-follower wait strategy.
17.4.4.3 Wait-on-Leader-Follower Wait Strategy

In the wait-on-leader-follower wait strategy (also known as “multithreaded reactive”), a client thread waits for its reply using the ORB’s leader-follower model, based on the Leader/Followers architectural pattern described in Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2), by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann.

The leader-follower model allows multiple threads to share a single reactor. One thread becomes the “leader” and runs the reactor’s event loop. The leader thread receives and processes the next event; other threads are “followers.” As soon as the leader thread receives an event, one of the follower threads becomes the new leader.

In the wait-on-leader-follower wait strategy, it is possible for a client thread waiting for a reply to be used by the ORB to handle incoming requests or other events. Thus, nested upcalls are possible in this configuration. When a client thread begins waiting for its reply, it is likely to become a follower in the leader-followers group of threads. It is awakened when one of the following occurs:

• Its reply is received.
• It is selected to become the leader and run the ORB’s reactor’s event loop.

If the reply is received before the waiting thread becomes the leader, the thread remains in the client role and does not have to temporarily switch to the server role. However, if the application is experiencing heavy load or an insufficient number of other threads are dedicated to the ORB (by calling CORBA::ORB::run() or CORBA::ORB::perform_work()), the probability that the waiting thread will temporarily switch to the server role is increased.

When the reply to the outbound request arrives, the thread waiting on the reply eventually breaks out of the leader-followers group and returns to its client role.

Motivation
The wait-on-leader-follower wait strategy is useful in the following situations:

• The application is either single-threaded or multithreaded and acts as both a client and a server.
• Nested upcalls are possible.
17.4 Multithreading in the Client

Configuration
The wait-on-leader-follower wait strategy is the default wait strategy used in TAO. It can be specified explicitly by supplying the following configuration option to the default client strategy factory:

\[
\text{static Client\_Strategy\_Factory "-ORBWaitStrategy mt"}
\]

See 22.3.4 for more information on using this option.

The wait-on-leader-follower wait strategy should be used with the thread-pool reactor type. Since that is the default reactor type in TAO, no additional configuration options are necessary to select it. See 17.3.3.3 for more information on the thread-pool reactor. See 20.7.6 for more information on options to configure the ORB’s reactor type.

Consequences
- In the wait-on-leader-follower wait strategy, it is possible for a client thread to be “stolen” by the ORB to handle inbound requests while waiting for a reply.
- A client thread should not hold a lock when invoking a request. If the thread is used to handle an inbound request while waiting for its reply, and the processing of that request results in an attempt to acquire the same lock, a deadlock could occur.
- Recursive nested upcalls may lead to unbounded stack growth, which will eventually crash the application.

For example, consider two processes, A and B, both using the wait-on-leader-follower wait strategy. Suppose A invokes a synchronous operation on B, which results in B invoking a synchronous operation on A, resulting in a nested upcall. If that nested upcall results in another invocation on B, which results in another invocation on A, and so on, one or both processes will eventually exhaust all available stack space.

17.4.4.4 Wait-on-Leader-Follower-No-Upcall Wait Strategy
The wait-on-leader-follower-no-upcall wait strategy is an experimental strategy that combines features of the wait-on-read and wait-on-leader-follower wait strategies. A client thread waits for its reply using the ORB’s leader-follower model, but no nested upcalls are permitted.
on the client thread while it waits for its reply. The waiting thread may handle non-upcall reactor events such as time-outs and connection-related events. Other threads operate normally.

**Note**  The wait-on-leader-follower-no-upcall wait strategy is an experimental wait strategy. It has not been exercised in a wide variety of use cases. Please use caution and due-diligence in testing your application’s behavior with this option if you decide to use it.

The wait-on-leader-follower-no-upcall wait strategy was motivated by the need to recognize connections opened in the client role and closed by the server. Using wait-on-read, the closure is not recognized until another invocation is made through that connection. A closed wait-on-read connection sits in a **CLOSE_WAIT** state, consuming a file descriptor.

With this strategy, a client thread waiting for a reply enters the leader-followers group of threads just as it does in the wait-on-leader-followers wait strategy. When the client thread becomes the leader, the reactor may ask it to handle a nested upcall. The client thread, however, defers the nested upcall back to the reactor for processing by another thread. The client thread does handle other reactor events such as connection establishment and time-out events.

In the wait-on-leader-follower-no-upcall wait strategy, it is not possible for a client thread waiting for a reply to be used by the ORB to handle incoming requests. In other words, nested upcalls are not possible. However, the client has more flexibility in its selection of connect strategy, transport multiplexing strategy, and flushing strategy.

**Motivation**
The wait-on-leader-follower-no-upcall wait strategy is useful in the following situations:

- The application is a hybrid client/server application with predominately server-side activity.
- The application manages connections that are opened by the client and closed by the server.
17.4 Multithreading in the Client

Note
Other use cases have not been thoroughly tested.

Configuration
The wait-on-leader-follower-no-upcall wait strategy can be specified by
supplying the following configuration option to the default client strategy
factory:

static Client_Strategy_Factory "-ORBWaitStrategy mt_noupcall"

See 22.3.4 for more information on using this option.

The wait-on-leader-follower-no-upcall wait strategy should only be used with
the thread-pool reactor type. Since that is the default reactor type in TAO, no
additional configuration options are necessary to select it. Note that no
warnings are given if one tries to use this option with an incompatible reactor
type. See 17.3.3.3 for more information on the thread-pool reactor. See 20.7.6
for more information on options to configure the ORB’s reactor type.

Consequences
• A client thread that is waiting for a reply cannot be “stolen” by the ORB
and used to process incoming requests; however, it can be used to handle
other reactor events, such as time-outs and connection-related events.

• A deadlock is possible. Because a client thread cannot be “stolen” by the
ORB to handle inbound requests while waiting for a reply, a
thread-pool-based server may “run out of threads” if all of its threads are
waiting for replies.

• The client thread is still called by the reactor to handle requests, but it
defers the requests back to the reactor. Therefore, a client that waits a long
time for a reply with lots of other inbound activity might result in
significantly increased CPU usage.

• In situations where a high percentage of threads are waiting for replies and
thus cannot handle inbound requests, it has been observed that the reactor
may fail to re-dispatch a request deferred to it by the waiting client thread.

• The wait-on-leader-follower-no-upcall wait strategy has not been
thoroughly tested with Asynchronous Method Invocation (AMI). The
client thread may not necessarily handle an AMI callback, but instead may
defer it to other threads.
Multithreading with TAO

- Unlike the wait-on-read strategy, connection establishment does not have to be blocking (i.e. the connect strategy does not have to be configured as blocked).

- Unlike the wait-on-read strategy, requests can be multiplexed across a single connection (i.e., the transport multiplexing strategy does not have to be configured as exclusive).

- Unlike the wait-on-read strategy, request messages do not have to be completely written to the transport (i.e., the flushing strategy does not have to be configured as blocking).

17.4.5 Optimizing Performance of Collocated Objects

When a client thread invokes an operation on a collocated CORBA object (i.e., an object that is implemented within the same address space as the client), the invocation may be optimized to bypass several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic (see 19.8.2).

Since collocated invocations do not follow the usual ORB request dispatching path, the server logic (servant implementation code) is executed on the same thread as the client invocation; that is, the client thread is “stolen” by the server. For example, even if an application is using the thread-pool concurrency model described in 17.3.3.2, a request invoked on a collocated object will not be subject to dispatching via the ORB’s thread pool.

Usually, the collocation optimization is desirable. Optimization of collocated invocations can result in dramatically increased performance and decreased latency compared to normal or ORB-mediated invocations. However, in the case of a hybrid server/client application, if the collocation optimization is allowed, a client thread making an invocation on a collocated object cannot be made available to the ORB for handling incoming client connections or requests, regardless of the effective wait strategy or other strategies that normally affect client thread behavior. If other threads in the process are not available to the ORB, remote client connection attempts or requests could be blocked until a thread becomes available.

Also, in the case of a real-time CORBA application, the client thread may not be running at the priority at which the request should be processed, possibly leading to priority inversions.
17.4 Multithreading in the Client

**Note** However, if the application is using RT CORBA, TAO uses a special real-time collocation resolver to determine whether the target object is collocated with the invoking thread. The real-time collocation resolver considers not only the address space and ORB of the target object, but also the POA, POA thread pool policy, thread pool id, thread pool lane, and effective priority model policy in making this determination. The result is that invocations on collocated objects in RT CORBA applications attempt to avoid priority inversions by not violating constraints imposed by priority and threading models. See Chapter 9 for more information on RT CORBA and 19.8.8 for information about an option to disable TAO’s real-time collocation resolution mechanism.

TAO provides an ORB initialization option to control the scope within which collocation optimizations are allowed. The `-ORBCollocation` option can have one of the following values:

- **global**
  Any object whose implementation resides in the same address space as the client is considered collocated. TAO uses global collocation by default.

- **per-orb**
  An object is considered collocated only if its implementation resides in the same address space as the client and the servant is accessed through the same ORB as that which the client is using to make the invocation. (This setting results in different behavior from global only in applications that use more than one ORB instance.)

- **no**
  The collocation optimization is disabled; invocations on objects implemented within the same address space as the client follow the same request dispatching path as invocations from remote clients.

See 19.8.2 for more information on using the `-ORBCollocation` option.

TAO also provides an ORB initialization option to specify the collocation strategy to use. Once the ORB has determined that the target of a request is collocated with the invoking client, the collocation strategy determines how the collocated invocation will be carried out. The `-ORBCollocationStrategy` can have one of the following values:

- **thru_poa**
Collocated invocations go through the POA and therefore respects the POA’s current state and request demultiplexing and processing policies. TAO uses the thru_poa collocation strategy by default.

- **direct**

Collocated invocations are carried out directly on the target servant, bypassing the POA completely. The servant must be in memory.

See 19.8.3 for more information on using the -ORBCollocationStrategy option.

## 17.5 Summary

Multithreading is important to many distributed applications, including applications that use CORBA. This chapter has discussed several aspects of multithreading with CORBA and TAO, including application programming interfaces, policies, configuration options, and design choices. Multithreading issues have been discussed from both server and client viewpoints.
Part 3

Run-time Configuration of TAO
CHAPTER 18

Configuring TAO Clients and Servers

18.1 Introduction

Though it is possible to use TAO out-of-the-box as a general purpose ORB, many applications will have special needs that are not met by TAO’s default configuration. For example, in some applications or operating environments, you may need to control the concurrency and locking strategies used when processing requests. In other cases, you may want to control how TAO behaves when a client invoking a request is collocated with the target object of the request. In addition, your operating environment may impose restrictions on your application’s use of resources such as memory, communication protocols, buffers, and endpoints.

TAO’s flexible and open architecture allows a high degree of run-time configuration for meeting the needs of a wide variety of applications and operating environments. TAO employs various design patterns to achieve this degree of flexibility.
18.1.1 Road Map
In this chapter, we introduce the concepts and techniques involved in configuring TAO at run time.

- In 18.2, we introduce the fundamental patterns, components, and techniques used for run-time configuration of TAO clients and servers.
- In 18.3, we introduce the ACE Service Configurator that TAO uses to dynamically configure its own components.
- In 18.4, we present command line options that control how TAO uses the service configurator.
- In 18.5, we described details of using the ACE Service Configurator framework.
- In 18.6, we describe the ACE XML Service Configurator.
- In 18.7, we discuss service objects, including how to create your own dynamically-loaded services.
- In 18.8, we discuss the ACE Service Manager, which allows remote administration of the service configurator.

To fully understand and take advantage of the ACE Service Configurator, read Chapter 5 of C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks (C++NPv2) and Chapter 19 of The ACE Programmer’s Guide (APG). See also the Component Configurator design pattern (75) in Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2).

18.2 Patterns and Components for Configuring TAO Clients and Servers
In this section, we describe the primary design patterns and components TAO uses for run-time configuration.

18.2.1 Factories
TAO relies on object factories to construct the required strategy and resource objects. An object factory is similar to a traditional factory in the following ways:
18.2 Patterns and Components for Configuring TAO Clients and Servers

- A traditional factory constructs and delivers products in response to the orders it receives, whereas an object factory constructs and delivers objects in response to requests.

- A traditional factory specializes in the products it constructs (e.g., an automobile factory may construct many varieties of cars and trucks, but will not construct bricks or children’s toys), whereas an object factory specializes in constructing the types of objects defined by its interface.

- A traditional factory uses supplied materials to construct products, whereas an object factory uses supplied parameters to construct objects.

18.2.2 Strategies
A strategy defines a means of obtaining a goal. In a server, for instance, one goal is to match incoming requests with the servants responsible for handling those requests; this process is called demultiplexing, or demux for short. TAO defines a number of request demultiplexing strategies for accomplishing this goal, each with its own idiosyncrasies and side effects. A server strategy factory is an object factory used in TAO that can produce a request demultiplexing strategy. Using dynamic configuration, it is possible to change at run time the way TAO demultiplexes requests to servants with no impact on application code.

18.2.3 Resources
A resource is a tool or source of supply, such as a buffer for containing the product of converting data from one format to another or a reactor used to detect the presence of new requests from the input source. TAO uses a resource factory to produce resources used by the ORB at run time.

18.2.4 TAO’s Default Strategy and Resource Factories
TAO organizes its strategies and resources into three groups, each group being served by one of the following default object factories:

- The Resource Factory controls creation of configurable resources used by the ORB core. Most of the resources required by the ORB core are fixed, but you have some flexibility in the choice of a reactor, the selection of communication protocols, and the behavior of CDR allocators.

- The Server Strategy Factory produces configurable elements utilized by the object adapter, such as request demultiplexing strategies.
• The Client Strategy Factory produces configurable elements that optimize object stub operations on the client side, such as concurrency within the client application, or multiple requests sharing a communication channel.

18.2.5 Specializing TAO’s Factories
Factories define interfaces for obtaining objects, and implementations supply the object instances. The default implementations supplied with TAO permit run time tuning via options set in the ACE Service Configurator. Sophisticated users, or those with special concerns about issues such as consistency or footprint, can tune at compile- or link-time by implementing specialized factories. The interfaces to the various configuration factories, and the options controlling the default implementations of these factories, are provided in subsequent chapters.

18.3 The ACE Service Configurator
The ACE Service Configurator is a framework that supports the creation of dynamically-configured applications. Applications may be comprised of different components, depending on commands—known as directives—provided to the framework. Directives processed by the service configurator direct it to load or unload service objects, start and stop the execution of these objects, and configure the state of statically-linked objects. By default, the service configurator reads directives from a file called svc.conf that it finds in the current directory. It is possible to declare alternative configuration files on the application’s command line, and even to supply directives directly to the service configurator. The service configurator provides additional services to the application, such as daemonizing a process or enabling the output of debug information from the ACE and TAO classes. Table 18-3 shows the ORB configuration options used to override the default behavior of the service configurator in TAO applications.

The pattern implemented by the ACE Service Configurator is described in detail in Chapter 5 of C++ Network Programming, Volume 2 (C++NPv2) and Chapter 19 of The ACE Programmer’s Guide.
18.3 The ACE Service Configurator

18.3.1 Using the static Directive

The service configurator provides a means for altering the configuration of an application at run time. Without shutting down the application, new service objects may be added dynamically from a library, and existing service objects may be removed. The directives most important for configuring TAO are those that supply initialization options to the statically-linked default factories. The static directive supplies initialization to service objects that are statically linked. The following line shows the form of a static directive:

```
static service_name "options"
```

For example, the following directive could be used to supply options to TAO’s statically-linked Resource_Factory initialization function at run time.

```
static Resource_Factory "options"
```

The components of the static directive are shown in Table 18-1.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>The service configurator directive. It must appear as shown.</td>
</tr>
<tr>
<td>service_name</td>
<td>The name by which the service object is registered with the service configurator. The name is Resource_Factory in the example above.</td>
</tr>
<tr>
<td>options</td>
<td>The list of options supplied to the service object’s initialization function. The options must be quoted if more than one whitespace-separated option is to be supplied to the service object.</td>
</tr>
</tbody>
</table>

All service objects are registered with the service configurator by name. This name is used to locate the object either to supply it with initialization parameters or (with other directives) to control its state.

TAO’s default factories register themselves with the service configurator by name as well. The default resource factory name is Resource_Factory, the default server strategy factory name is Server_Strategy_Factory, and the default client strategy factory name is Client_Strategy_Factory. You supply initialization options to these factories using the following directives:
static Resource_Factory "-ORBoption value ..."
static Client_Strategy_Factory "-ORBoption value ..."
static Server_Strategy_Factory "-ORBoption value ..."

**Note** When you use the default configuration factories, you only need to supply directives to the service configurator if you want to apply option values other than the defaults. The default configuration factories properly initialize without the use of directives.

### 18.3.2 Using the *dynamic* Directive

When developing an application, using the dynamic configuration capabilities of the service configurator, you use the *dynamic* directive to trigger the loading and initialization of service objects from libraries. The following line shows the form of the *dynamic* directive:

```
dynamic service_name base_object_type library:factory_function() "options"
```

For example, to replace the default resource factory with one of your own defined in libmylib.so (on Unix) or mylib.dll (on Win32), the directive is:

```
dynamic Resource_Factory Service_Object * mylib:make_resources() "param ..."
```

The components of the *dynamic* directive are shown in Table 18-2.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dynamic</code></td>
<td>The service configurator directive. It must appear as shown.</td>
</tr>
<tr>
<td><code>service_name</code></td>
<td>The name used to register the newly-created service object with the service configurator. In the example above, the name is Resource_Factory.</td>
</tr>
<tr>
<td><code>base_object_type</code></td>
<td>The basic type returned by the factory function. For general service objects, use Service_Object*.</td>
</tr>
</tbody>
</table>
TAO does not require the use of any dynamically-loaded service objects. However, it is possible that you may use them because of application requirements, or merely personal or corporate preference. For example, the advanced resource factory, which allows configuration of advanced options, is often loaded dynamically. (See 20.5 for information on using the advanced resource factory.) You may also want to use alternative communications protocols. These alternative protocols may be supplied by creating new pluggable protocol factories, then loading them as dynamic service objects. The use of alternative protocols is explored in greater detail in Chapter 15.

### 18.4 Service Configurator Control Options

Table 18-3 lists options that influence the behavior of the service configurator used in TAO. These options duplicate many of the common options available to applications built using the Service Configurator pattern. If a client or server application is based on this pattern, these options are redundant.

#### Table 18-3 Service Configurator ORB Initialization Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBDaemon</td>
<td>19.8.4</td>
<td>Instruct the process to run as a daemon.</td>
</tr>
</tbody>
</table>
18.5 The ACE Service Configurator Framework

This section presents details of the ACE Service Configurator framework. It is not necessary to explicitly use this framework when developing TAO-based applications. However, the framework is useful when developing modular applications that can benefit from dynamic configuration.

The ACE Service Configurator framework allows decisions about system configuration to be deferred until the deployment phase of a project, rather than during the design phase. Deployment personnel configure service objects into processes and assemble a running system. Processes may contain many of these service objects, or the service objects may be spread across several processing elements; the choice is made at deployment time. Service objects may be active (running in their own thread) or reactive (registering with a reactor to handle events), and a process may contain any combination thereof.

To successfully deploy a system based on the service configurator framework, the following two fundamental goals must be achieved:

- Service objects must fit in to the framework (see 18.7).
- Service objects must be loaded into the system (see 18.5.1).

### 18.5.1 Loading Service Objects

The service configurator loads services for you by initializing an `ACE_Service_Config` object. Arguments are passed to the service configurator using an array of strings, similar to the way command line arguments are passed to the main function of an application. A very common use case for the service configurator is to simply pass the `argc` and `argv` parameters of `main()` to the service configurator’s `open()` function. The

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ORBSvcConf config_file_name</code></td>
<td>19.8.32</td>
<td>Specify an alternate service configurator file name.</td>
</tr>
<tr>
<td><code>-ORBSvcConfDirective directive</code></td>
<td>19.8.33</td>
<td>Pass a single directive directly to the service configurator.</td>
</tr>
<tr>
<td><code>-ORBServiceConfigLoggerKey logger_key</code></td>
<td>19.8.28</td>
<td>Specify where to write ORB logging output.</td>
</tr>
</tbody>
</table>
example below shows a generic main program that loads services, then runs them:

```c
#include <ace/Service_Config.h>

int main(int argc, char** argv)
{
    // open() reads svc.conf, creates services, initializes
    ACE_Service_Config configurator;
    configurator.open(argc,argv);

    // run event loop to handle client requests
    ACE_Reactor::instance()->run_event_loop();

    // a multithreaded program can use
    // ACE_Thread_Manager::instance()->wait();
}
```

In the above example, the service configurator is opened, using the program’s command line arguments as initialization parameters. The service configurator locates a file containing directives, then loads and initializes service objects as indicated by these directives. In 18.4, we describe the ORB initialization options that provide some control over the service configurator. Table 18-4 shows the arguments processed by the service configurator in its `open()` function.

**Table 18-4 Service Configurator Command Line Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f file</td>
<td>Supply a file containing directives other than the default <code>svc.conf</code>.</td>
</tr>
<tr>
<td>-s signal</td>
<td>Supply an alternative signal for causing the service configurator to reprocess the directives file. By default, SIGHUP is used.</td>
</tr>
<tr>
<td>-d</td>
<td>Turn on debugging in ACE and TAO objects.</td>
</tr>
<tr>
<td>-n</td>
<td>Ignore static directives.</td>
</tr>
<tr>
<td>-y</td>
<td>Process static directives. The default behavior. It overrides -n.</td>
</tr>
<tr>
<td>-b</td>
<td>Turn the containing process into a daemon.</td>
</tr>
<tr>
<td>-S directive</td>
<td>Supply a directive to the service configurator directly. The -S argument may be repeated to supply multiple directives to the service configurator.</td>
</tr>
<tr>
<td>-k logger_key</td>
<td>Specify where to write ORB logging output.</td>
</tr>
</tbody>
</table>
Configuring TAO Clients and Servers

Synopsis

In addition to the open() function, the service configurator provides some additional functions as part of a control interface. Though not complete, the class definition below shows the useful functions available to application developers. Other functions in the public interface to ACE_Service_Config are for use by other elements of the framework:

class ACE_Export ACE_Service_Config
{
public:
    static int open (int argc,
        ACE_TCHAR *argv[],
        const ACE_TCHAR *logger_key = ACE_DEFAULT_LOGGER_KEY,
        int ignore_static_svcs = 1,
        int ignore_default_svc_conf_file = 0,
        int ignore_debug_flag = 0);
    static int open (const ACE_TCHAR program_name[],
        const ACE_TCHAR *logger_key = ACE_DEFAULT_LOGGER_KEY,
        int ignore_static_svcs = 1,
        int ignore_default_svc_conf_file = 0,
        int ignore_debug_flag = 0);
    static int close (void);
    static void reconfigure (void);
    static int process_directive (const ACE_TCHAR directive[]);
}

18.5.2 Opening and Closing the Service Configurator

There are two forms of the open() function. The first accepts a list of command line arguments. The second accepts only a program name. For both forms, the remaining parameters are optional. Table 18-5 shows the arguments used by these two functions.

Table 18-5 ACE_Service_Config::open() Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>int</td>
<td>Number of arguments in the argv list. Usually the same value supplied as argv to main().</td>
</tr>
<tr>
<td>argv</td>
<td>ACE_TCHAR*[]</td>
<td>List of command line arguments. Usually the same value supplied as argv to main(). argv[0] is usually the program name.</td>
</tr>
</tbody>
</table>
18.5 The ACE Service Configurator Framework

Calling `ACE_Service_Config::close()` shuts down all services and deletes all memory allocated by the service configurator.

### 18.5.3 Commanding the Service Configurator

The remaining `ACE_Service_Config` functions are used by an application to command the service configurator. These functions provide internal access to the service configurator similar to the external interface available through the use of signals and service configuration files. Another way to command the service configurator is to use the service manager (see 18.8). The service manager uses these functions among others to allow remote configuration of the service configurator.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>program_name</code></td>
<td>const ACE_TCHAR[]</td>
<td>Alternate first argument: used when the command line is not to be interpreted by the service configurator.</td>
</tr>
<tr>
<td><code>logger_key</code></td>
<td>const ACE_TCHAR*</td>
<td>Rendezvous point for connection to the ACE Logger daemon. Default depends on platform type (platforms that support stream pipes use /tmp/server_daemon. Others use localhost:20012).</td>
</tr>
<tr>
<td><code>ignore_static_svcs</code></td>
<td>int</td>
<td>Indicates that the service configurator should not process any static services. Default is 1 (do not process static services).</td>
</tr>
<tr>
<td><code>ignore_default_svc_conf_file</code></td>
<td>int</td>
<td>Indicates the service configurator should not read the default service configuration file, svc.conf, even if no other file is specified using the -f command line option. Default is 0 (attempt to read from svc.conf unless overridden).</td>
</tr>
<tr>
<td><code>ignore_debug_flag</code></td>
<td>int</td>
<td>Indicates that the service configurator should ignore any debug settings. If non-zero, the application is responsible for setting ACE_Log_Msg::priority_mask appropriately. Default is 0 (do not ignore debug settings).</td>
</tr>
</tbody>
</table>
The following function instructs the service configurator to reprocess all directives both in service configuration files and on the command line:

```c
static void reconfigure (void);
```

Calling `ACE_Service_Config::reconfigure()` is the same as signalling the process by using whatever signal the service configurator is set to monitor (e.g., `SIGHUP`).

You may provide a single directive to the service configurator using:

```c
static int processDirective (const ACE_TCHAR directive[]);
```

### 18.5.4 Additional Directives

In addition to the static and dynamic directives introduced in 18.3, the service configurator also processes the directives `remove`, `suspend`, and `resume`.

**The remove Directive**

Part of the dynamic nature of the service configurator is the ability to remove a currently-loaded service object. The `remove` directive causes the service configurator to call the object’s finalizer, then remove the instance of the object from the repository. Typically, this would not be done during process start-up, but might be part of a service configuration file. For example, you might modify the service configuration file by adding a `remove` directive, the send a `SIGHUP` signal to the process to force the service configurator to re-read the file and process the directives therein. Another possibility is to use the service manager to process the directive remotely (see 18.8).

The form of the `remove` directive is:

```plaintext
remove service_name
```

For example, to remove a service previously loaded, using the name `my_service`, the directive is:

```plaintext
remove my_service
```
The **suspend** Directive
A running service may be halted for a period of time, without removing the service from the process, by suspending the service. A suspended service may be resumed later. The **suspend** directive causes the service configurator to call the `suspend()` function on the associated service object if the service object is not already suspended.

The form of the **suspend** directive is:

```
suspend service_name
```

For example, to suspend a running service named `my_service`, the directive is:

```
suspend my_service
```

The **resume** Directive
A suspended service may be resumed by placing the **resume** directive in the service configuration file, then triggering the reprocessing of the configuration file via a signal. The ability to suspend and resume services is useful for remote administration of the service configurator. You can also pass the **resume** directive remotely using the service manager (see 18.8).

The form of the **resume** directive is:

```
resume service_name
```

For example, to resume a suspended service named `my_service`, the directive is:

```
resume my_service
```

### 18.6 XML Service Configurator

By default, ACE uses the **classic** service configurator (described in 18.5). However, an XML front end to the service configurator is also provided. It can be enabled by defining the `ACE_HAS_XML_SVC_CONF` macro in `$ACE_ROOT/ace/config.h`. When this macro is enabled, the ACE library will be built with support for the XML service configurator and will not
support the classic service configurator format. A script, $ACE_ROOT/bin/svcconf-convert.pl, is provided that will convert the classic service configurator files into XML-based service configurator files.

18.6.1 Service Configurator DTD

The file $ACE_ROOT/ACEXML/apps/svcconf/svcconf.dtd contains the Service Configurator DTD. It is shown here for convenience:

```xml
<!-- $Id: svcconf.dtd,v 1.1.1.1 2005/01/03 19:35:39 chad Exp $   -->
<!-- Document Type Definition for XML ACE Service Config files -->
<!-- An ACE_Svc_Conf document contains zero or more entries -->
<!-- The entries are processed in the order they appear -->
<!-- in the ACE_Svc_Conf file. -->
<!ELEMENT ACE_Svc_Conf (dynamic|static|suspend|resume|remove|stream|streamdef)*>
<!-- Streams are separate into two elements. One defines how -->
<!-- the stream should be constructed and the other defines -->
<!-- what to do with it. The identity of a stream is defined -->
<!-- in the first dynamic/static element. -->
<!ELEMENT streamdef ((dynamic|static),module)>
<!-- Do we ever need to suspend/resume/remove modules when -->
<!-- constructing a stream? Should we leave only dynamic -->
<!-- and static here? -->
<!ELEMENT module (dynamic|static|suspend|resume|remove)+>
<!-- A 'stream' element controls the stream object -->
<!ELEMENT stream (module)>
<!ATTLIST stream id IDREF #REQUIRED>
<!-- A 'dynamic' entry. -->
<!ELEMENT dynamic (initializer)>
<!ATTLIST dynamic id ID #REQUIRED
status (active|inactive) "active"
type (module|service_object|stream) #REQUIRED>
<!-- The kind of attributes the corresponding initializer -->
<!-- should take seems to be determined by the 'type' -->
<!-- attribute. Should we further partition the dynamic -->
<!-- element definition into several elements? E.g. into -->
<!ELEMENT dyn_service_object/dyn_module/dyn_stream? -->
<!-- Nanbor: Will that be too confusing? -->
<!ELEMENT dynamic (initializer)>
<!ATTLIST dynamic id ID #REQUIRED
status (active|inactive) "active"
type (module|service_object|stream) #REQUIRED>
<!-- Initializing function for dynamic entry. -->
<!ELEMENT initializer EMPTY>
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
18.6 XML Service Configurator

18.6.2 XML Service Configurator Syntax
In this section, we discuss the dynamic, static, remove, suspend, and resume entries of the XML configurator syntax.

18.6.2.1 Dynamic
The dynamic example discussed in 18.3.2 is as follows:

dynamic Resource_Factory Service_Object * mylib:_make_resources() "param ...

The equivalent in XML format is shown below:

<dynamic id="Resource_Factory" type="Service_Object">
  <initializer path="mylib" init="_make_resources()" params="param ...
</dynamic>

18.6.2.2 Static
The static example discussed in 18.3.1 is as follows:

static Resource_Factory "-ORBoption value ..."
The equivalent in XML format is shown below:

```
<static id="Resource_Factory" params="#ORBOption value ..."/>
```

18.6.2.3 Remove
The remove example discussed in 18.5.4 is as follows:

```
remove my_service
```

The equivalent in XML format is shown below:

```
<remove id="my_service"/>
```

18.6.2.4 Suspend
The suspend example discussed in 18.5.4 is as follows:

```
suspend my_service
```

The equivalent in XML format is shown below:

```
<suspend id="my_service"/>
```

18.6.2.5 Resume
The resume example discussed in 18.5.4 is as follows:

```
resume my_service
```

The equivalent in XML format is shown below:

```
<resume id="my_service"/>
```

18.7 Service Objects
The class ACE_Service_Object is the base class for all objects that can be loaded by the service configurator. Class ACE_Service_Object inherits from both ACE_Shared_Object and ACE_Event_Handler, as shown in
18.7 Service Objects

Figure 18-1. These base classes provide the behavior required to catalog the service object in a repository, and to provide basic interaction with a reactor.

<table>
<thead>
<tr>
<th>ACE_Shared_Object</th>
<th>ACE_Event_Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>init()</td>
<td>handle_input()</td>
</tr>
<tr>
<td>fini()</td>
<td>handle_output()</td>
</tr>
<tr>
<td>info()</td>
<td>handle_timeout()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACE_Service_Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>suspend()</td>
</tr>
<tr>
<td>resume()</td>
</tr>
</tbody>
</table>

Figure 18-1  ACE Service Object Inheritance

18.7.1 Interface Definition

All of the virtual functions of a service object’s interface have default *no-op* implementations. The class definition below shows the interface to the service object, including some of the inherited elements:

```cpp
Synopsis
class ACE_Export ACE_Service_Object :
    public ACE_Event_Handler, public ACE_Shared_Object {
    public:
        ACE_Service_Object (ACE_Reactor * = 0);
        virtual ~ACE_Service_Object (void);
        virtual int suspend (void);
        virtual int resume (void);

        // From ACE_Shared_Object
        virtual int init (int argc, ACE_TCHAR *argv[]);
        virtual int fini (void);
        virtual int info (ACE_TCHAR **info_string, size_t length = 0) const;

        // From ACE_Event_Handler
        virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);
        virtual int handle_output (ACE_HANDLE fd = ACE_INVALID_HANDLE);
    }
```
virtual int handle_exception (ACE_HANDLE fd = ACE_INVALID_HANDLE);
virtual int handle_timeout (const ACE_Time_Value &current_time,
    const void *act = 0);
virtual int handle_close (ACE_HANDLE handle, ACE_Reactor_Mask close_mask);
};

The functions shown above are examined in the following subsections.

### 18.7.2 Service Initialization and Finalization

Dynamic loading and discarding of service objects occurs throughout a program’s lifetime. Externally, this is facilitated by using the dynamic and remove directives. A dynamically-loaded object is initialized through a call to its init() function. When dynamically loading an object, the service configurator first locates and opens the shared library that contains the object’s factory function. Then the service configurator invokes the function, creating a new instance of the object. Finally, the service configurator calls the new object’s init() function, supplying the initialization parameters from the service configuration directive. The initialization signature is:

```c
virtual int init (int argc, ACE_TCHAR *argv[]);
```

As shown above, init() takes arguments supplied with the dynamic directive in an array as parameters, similar to the program’s main() function. Statically-loaded objects are also initialized through a call to init(), using arguments supplied with a static directive. If initialization is successful, init() returns zero. The function returns -1 if initialization could not complete.

The service configurator calls the finalizer, not the destructor, when processing a remove directive. The destructor is called for dynamic objects only if they are resident in the process when the process is terminated. The finalizer signature is:

```c
virtual int fini (void);
```

Although fini() returns a result code, the service configurator framework does not currently check the value.
18.7 Service Objects

18.7.3 Service Creation

The service configurator creates new objects by invoking factory functions. A conforming factory function may have any name, but it must conform to the following interface:

```c
typedef void (*ACE_Service_Object_Exterminator)(void *);
ACE_Service_Object * factory (ACE_Service_Object_Exterminator *gobbler);
```

The factory must return a pointer to a (typically new) instance of the service object type in question, and must also initialize the `gobbler` pointer to point to a function that conforms to the `ACE_Service_Object_Exterminator` function interface. This exterminator function is called when it is appropriate to delete the object.

ACE_Static_Svc_Descriptor

For statically-linked service objects, it is the application’s responsibility to create an instance of the object and to register the object with the service configurator. To do this, a factory function, as shown above, must still be used, and an object of type `ACE_Static_Svc_Descriptor` must be created and provided to the service configurator.

The `ACE_Static_Svc_Descriptor` class shown below is a wrapper around all the information needed by the service configurator to use static service objects:

```cpp
class ACE_Static_Svc_Descriptor
{
    public:
        const ACE_TCHAR *name_;  
        int type_; 
        ACE_SERVICE_ALLOCATOR alloc_; 
        u_int flags_; 
        int active_; 
        void dump (void) const; 
        int operator== (ACE_Static_Svc_Descriptor &) const; 
        int operator!= (ACE_Static_Svc_Descriptor &) const; 
};
```

Since `ACE_Static_Svc_Descriptor` attributes are publicly accessible, no constructor is needed, other than the default. These attributes are described below:
Configuring TAO Clients and Servers

- **name_** is a string representing the unique identifier used to register the service with the service configurator. Note that the type is declared as a const ACE_TCHAR * rather than a const char *. The macro ACE_TCHAR enables building on platforms requiring wide character strings.

- **type_** is a value indicating the type of wrapper used by the service configurator. For service objects, use the macro ACE_SVC_OBJ_T to initialize the type.

- **alloc_** is a pointer to the factory function used to create an instance of the service object. This function must return a pointer to the appropriate type of object, usually a pointer to an ACE_Service_Object. As with other “boilerplate” functions used in conjunction with the service configurator, there is a macro defined that becomes a service object factory function. The name of the factory function is represented by the macro ACE_SVC_NAME(service_object_class_name).

- **flags_** is used to communicate destruction semantics for the service object and its container. There are two values used to set flags_, DELETE_OBJ and DELETE_THIS, defined in the class ACE_Service_Type. DELETE_THIS instructs the service configurator to remove the container of the service object when the service is shut down. DELETE_OBJ instructs the service configurator to delete the service object as well by invoking the exterminator supplied by the factory function, after calling fini(). Typically, the container should always be deleted by specifying DELETE_THIS, and if a service object is created in the factory function, it should be deleted as well by specifying DELETE_OBJ. These values may be combined via bitwise OR for assignment to flags_, as in:

  \[
  \text{ACE\_Service\_Type::DELETE\_THIS | ACE\_Service\_Type::DELETE\_OBJ}
  \]

- **active_** is set to a non-zero value to indicate that the service should be run in a separate thread by having its activate() function called or being registered with the reactor.

### Helper Macros
ACE defines macros in $ACE_ROOT/ace/Global_Macros.h to simplify the creation of factory and exterminator functions, as well as static object descriptors.
18.7 Service Objects

Use the following macros in your service object’s class header file to declare the static functions and classes used to create and manage service objects:

- **ACE_STATIC_SVC_DECLARE_EXPORT**(EXPORT_NAME, SVC_CLASS) must be included when your service is to be statically linked with your application. This macro declares a class with a constructor that will ensure an instance of your service is created. If necessary (e.g., in Windows DLLs), the name of the class is exported for visibility outside the library in which it is defined. The EXPORT_NAME parameter is the prefix for the export directive ("_Export" will automatically be appended to this prefix). The export macro has no effect on platforms where it is not needed. The SVC_CLASS parameter is the name of your service object class.

**Note**  
_A header file defining the export macro can be automatically generated using the Perl script $ACE_ROOT/bin/generate_export_file.pl. The script provides instructions for using it._

- **ACE_STATIC_SVC_DECLARE**(SVC_CLASS) is similar to **ACE_STATIC_SVC_DECLARE_EXPORT**, except that it does not include the macro to export the symbol. You can use this macro on platforms that do not need to export symbols from DLLs. The SVC_CLASS parameter is the name of your service object class.

- **ACE_FACTORY_DECLARE**(EXPORT_NAME, SVC_CLASS) declares the uniquely-named factory function that the service configurator uses to create an instance of your service class. The parameters are the same as for **ACE_STATIC_SVC_DECLARE_EXPORT**.

The following macros are used in the source files to complete the definition of functions and classes that are declared by the above macros:

- **ACE_STATIC_SVC_DEFINE**(SVC_CLASS, NAME, TYPE, FN, FLAGS, ACTIVE) is used to statically initialize an ACE_Static_Svc_Descriptor. The SVC_CLASS parameter is the name of your service object class. The remaining parameters are described as the fields of the ACE_Static_Svc_Descriptor structure.

- **ACE_FACTORY_DEFINE**(EXPORT_NAME, SVC_CLASS) defines the body of the factory function declared with **ACE_FACTORY_DECLARE**. This factory function creates an instance of your service object and returns a
pointer to it as a pointer to an ACE_Service_Object. ACE_FACTORY_DEFINE also defines a function to clean up your service object. The parameters to this macro are the same as for ACE_FACTORY_DECLARE.

- ACE_SVC_NAME(SVC_CLASS) expands to the name of the service object factory function. This macro is useful when supplying to the service configurator a pointer to the factory function.

- ACE_STATIC_SVC_REQUIRE(SVC_CLASS) automatically registers your service with the service configurator by adding to the service configurator’s service repository the service descriptor created with the ACE_STATIC_SVC_DEFINE macro. This macro also creates a static instance of your service class to ensure that the service is registered before main().

### 18.7.3.1 Dynamic Service Example

The following example shows a service object that can be loaded dynamically. The code compiles to a shared library that is then loaded when the service configurator processes the dynamic directive for this shared object:

```cpp
//declare the service
class My_Service: public ACE_Service_Object
{
    public:
        int init (int argc, ACE_TCHAR *argv[]);
        int fini (void);
};

//declare the service factory
ACE_FACTORY_DECLARE (My_Lib, My_Service);

// define the service
int My_Service::init (int argc, ACE_TCHAR *argv[]) {
    // parse the args and get things initialized
    return 0;
}

int My_Service::fini (void) {
    // service is shut down
    return 0;
}

// define the factory
ACE_FACTORY_DEFINE (My_Lib, My_Service)
```
18.7.3.2 Static Service Example

The following example shows the modifications (in **boldface**) from the previous example required to statically instantiate an object and register it with the service configurator when the service is to be statically linked to the application.

```cpp
//declare the service
class My_Service: public ACE_Service_Object
{
    public:
        int init (int argc, ACE_TCHAR *argv[]);
        int fini (void);
};

//declare the static service descriptor
ACE_STATIC_SVC_DECLARE_EXPORT (My_Lib, My_Service);

//declare the service factory
ACE_FACTORY_DECLARE (My_Lib, My_Service);

// define the service
int My_Service::init (int argc, ACE_TCHAR *argv[]) {
    // parse the args and get things initialized
    return 0;
}

int My_Service::fini (void) {
    // service is shut down
    return 0;
}

// define the static service descriptor
ACE_STATIC_SVC_DEFINE (
    My_Service,                     // our service class
    "My_Static_Service",            // the name used to register the service
    ACE_SVC_OBJ_T,                  // use the service object container
    &ACE_SVC_NAME(My_Service),      // a reference to the factory function
    ACE_Service_Type::DELETE_THIS,  // delete the container when done
    ACE_Service_Type::DELETE_OBJ,   // delete the service object when done
    0);                             // this object is not active

// define the factory
ACE_FACTORY_DEFINE (My_Lib, My_Service);

// register the service descriptor with the service configurator
ACE_STATIC_SVC_REQUIRE(My_Service);
```
18.7.4 Service Information

Services are expected to provide information about themselves by overloading the info() function. The signature of the info() function is:

```cpp
virtual int info (ACE_TCHAR **info_string, size_t length = 0) const;
```

The arguments are a pointer to a string buffer and the buffer’s length. If the provided info_string is a null pointer, the info() function creates a string buffer. Otherwise, info() copies the information to that buffer using the provided length as a limit. The result should be the length of the contents in the buffer, or -1 if there is an error.

18.7.5 Service State

Services may be suspended for a time, then later resumed. To support this behavior, the service object must overload the following two functions:

```cpp
virtual int suspend (void);
virtual int resume (void);
```

The service configurator calls suspend() when it processes a suspend directive. The service object is expected to enter into a state in which it does not respond to inputs. For reactor-based services, this usually means unregistering from the reactor. For active services, the service suspends its associated thread. A suspended service does not have to finalize itself, as it will stay resident in the process.

You reactivate a suspended service with resume(), called by the service configurator when it processes a resume directive. A suspended service should continue to operate normally after being resumed.

18.7.6 Obtaining Services

When building a dynamically-configured application, it is useful to be able to locate services by name in a manner similar to using the Naming Service. The service configurator framework provides this capability by using the following template for looking up service objects of a particular type by name:

**Synopsis**

```cpp
template <class TYPE>
class ACE_Dynamic_Service : public ACE_Dynamic_Service_Base
{
public:
    static TYPE* instance (const ACE_TCHAR *name);
};
```
The function `instance()` will return a pointer to a service object of the appropriate type registered with the service configurator, using the specified name. If no object with that name is found, then `instance()` will return a null pointer.

Care must be taken that the service object associated with the name is really of the specified type. Due to the lack of run-time type safety in some C++ environments, incorrect casting can occur.

The following example uses `ACE_Dynamic_Service` to obtain a pointer to a `My_Service` object:

```cpp
My_Service* svc_ptr;
svc_ptr = ACE_Dynamic_Service<My_Service>::instance("my_service");
```

## 18.8 ACE Service Manager

The service manager is a service object that is statically linked with ACE. It registers itself with the service configurator using the name `ACE_Service_Manager`. The service manager allows users to control the service configurator via a telnet connection. Using the service manager, it is possible to send remotely any directive accepted by the service configurator. Other capabilities include signaling the service configurator to reprocess its service configurator file, and getting a list of available service objects. The service manager is configured by the service configurator. Table 18-6 shows the options that may be supplied to the service manager.

### Table 18-6 ACE Service Manager Command Line Arguments

<table>
<thead>
<tr>
<th>Option</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d</code></td>
<td>Set the service manager to debug mode.</td>
</tr>
<tr>
<td><code>-p portno</code></td>
<td>Specify the port number for listening. Default is 10000.</td>
</tr>
<tr>
<td><code>-s signal</code></td>
<td>Specify an alternative signal to trigger a reconfiguration. Default is SIGHUP.</td>
</tr>
</tbody>
</table>

The following example shows how to supply these options to the service manager using the static directive:
When initialized, the service manager will listen on the specified TCP port for incoming connections. When a connection is made, a single command is accepted from the client, the result (if any) is returned, and the connection is closed. Table 18-7 shows the commands the service manager will accept and its action in response (commands are case sensitive).

Table 18-7 ACE Service Manager Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>directive</td>
<td>Takes any single service configurator directive (dynamic, suspend, resume, etc.) and processes it using ACE_Service_Config::process_directive().</td>
</tr>
<tr>
<td>help</td>
<td>Iterate through the list of service objects registered with the service configurator and call the info() function for each. Returns the collection of object names and information strings.</td>
</tr>
<tr>
<td>reconfigure</td>
<td>Causes the service configurator to reprocess the directives in the service configuration file.</td>
</tr>
</tbody>
</table>

The following example shows how to send commands/directives to the service manager in a process running on a host named malory listening on port 3911:

```
$ echo help | telnet malory 3911
$ echo reconfigure | telnet malory 3911
$ echo "suspend my_service" | telnet malory 3911
```

18.9 Summary

The ACE Service Configurator framework provides a mechanism for designing systems composed of collections of services and to defer decisions about service configuration until system deployment. Further, services may be dynamically configured, meaning they may be loaded or unloaded at any time during the life of the host process.
CHAPTER 19

ORB Initialization Options

19.1 Introduction

Many of the common behaviors of the ORB can be controlled at run time by passing options to the ORB initialization function, CORBA::ORB_init(). ORB initialization options are commonly passed into the program from the command line, using the argc and argv parameters made popular by C and UNIX. However, you can also create and pass these ORB initialization options programmatically.

You can use ORB initialization options to control the following ORB behaviors:

- Service configurator behavior.
- Quantity of debugging information output.
- Optimizations applied during request transfer and processing.
- Connection management and protocol selection behavior.
- Use of the Implementation Repository.

All of the ORB initialization command line options are of the form:
ORB Initialization Options

-ORBOption [arguments...]

Some examples are:

-ORBDebug
-ORBRCvSock 1024
-ORBListenEndpoints iio://localhost:9999

You can pass several options on a single command line, for example:

./server -ORBDebug -ORBCollocation global -ORBCollocationStrategy thru_poa

In addition, you can supply some options more than once on the same command line. Most ORB initialization options apply to both clients and servers.

The initialization function parses these options in a case-insensitive manner (e.g., -ORBCollocationStrategy and -ORBcollocationstrategy are equivalent), removes them from the argv list, then decrements argc. Since the command line may also contain options that are specific to your application, you may find it easier to defer your own application-specific command line processing until after calling CORBA::ORB_init().

This chapter explains all of the ORB initialization options and describes the appropriate context for their use.

Note The ORBId, ORBInitRef, ORBDefaultInitRef, ORBListenEndpoints, ORBNoProprietaryActivation, and ORBServerId options described in this chapter are part of the standard CORBA specifications. All other options are TAO-specific.

19.2 Interface Definition

The interface for CORBA::ORB_init() is defined by the OMG CORBA specification as follows:

// C++
namespace CORBA { 
    static ORB_ptr ORB_init(
        int& argc, char** argv, const char* orb_identifier = "");

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University.
Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
The first two arguments to CORBA::ORB_init() are the familiar argc and argv parameters popularized by C and UNIX for passing command line arguments to main(). You may frequently pass command line arguments directly from main() to CORBA::ORB_init() via argc and argv. The third argument is an ORB identifier that defaults to an empty string. The first time a particular ORB identifier is used within a process as an argument to CORBA::ORB_init(), an ORB with that name will be created. Subsequent calls within the same process, using this same identifier, will return a reference to the previously-created ORB. You can create more than one ORB instance in a process by calling CORBA::ORB_init() multiple times, using a different orb_identifier value each time.

An alternate method for passing orb_identifier to CORBA::ORB_init() is to pass it in the argv list as the argument for the -ORBId option (i.e., -ORBId orb_identifier). If both methods are used, a directly-passed non-empty value will take precedence over the value of the -ORBId option. In either case, if orb_identifier refers to a previously-created ORB, all other arguments passed via the argv list will be ignored because the ORB has already been initialized.

As stated previously, CORBA::ORB_init() parses arguments from the argv list and “consumes” any arguments it recognizes that begin with -ORB. This means the value of argc and the contents of argv may be modified by CORBA::ORB_init(). If you want to preserve command line arguments, you should make a copy of the argv list before calling CORBA::ORB_init().

An alternative to using the command line arguments is to construct your own argument list programmatically (e.g., to pass a unique set of arguments to each ORB instance in your application).

---

**Note**

The TAO implementation of CORBA::ORB_init() assumes that argv[0] contains the program name and begins parsing at argv[1]. So, if you construct your own argument list, be sure to provide a “dummy” argument for argv[0].

---

Here is an example showing how to create your own argument list for CORBA::ORB_init() programmatically:
ORB Initialization Options

// Set up argv[] array of -ORB options
char* argv[] = {
    "dummy",                   // argv[0] is skipped
    ":ORBId", "MyORB",         // Provide a unique ORB id
    ":ORBDebug",               // Enable debug messages
    ":ORBDebugLevel", "6",     // Set debug level
    ":ORBListenEndpoints",
    "iiop://myhost:9999",      // Specify ORB's listening endpoint
    0                          // argv[] should end with a null value
};

// Set value of argc based on actual contents of argv[]
int argc = (sizeof(argv)/sizeof(char*)) - 1;

// Pass the arguments to CORBA::ORB_init()
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

The first time you call CORBA::ORB_init() with a given ORB identifier, a new instance of the class TAO_ORB_Core is created. TAO_ORB_Core is the TAO implementation class for the CORBA::ORB interface. You will find its class definition in $TAO_ROOT/tao/ORB_Core.h. Each TAO_ORB_Core instance encapsulates several resources and components used by the ORB core, such as an acceptor, a root POA, a resource factory, a server strategy factory, and a client strategy factory. The resource factory is discussed in Chapter 20, the server strategy factory in Chapter 21, and the client strategy factory in Chapter 22.

Since each call to CORBA::ORB_init(), using a unique ORB identifier, creates a new TAO_ORB_Core instance, implementing an ORB-per-thread concurrency model is as simple as calling CORBA::ORB_init() once from each thread, making sure to supply a unique ORB identifier for each call. For example, the thread identifier could be used to generate a unique ORB identifier. In $TAO_ROOT/performance-tests/Cubit/TAO/MT_Cubit, you will find sample code for creating an application with an ORB-per-thread concurrency model.

Since each ORB core instance encapsulates an acceptor, each ORB you create will be listening on one or more endpoints. These endpoints may be specified using the -ORBEndpoint or -ORBListenEndpoints option or with the TAO_ORBENDPOINT environment variable. Otherwise, the ORB creates an endpoint for each protocol known to the resource factory. The default configuration of the resource factory causes the ORB to create only an IIOP endpoint. For more details on endpoints, see the discussion of the
-ORBEndpoint option in 19.8.10. For more details on the supported protocols, see the discussions of TAO’s pluggable protocols in Chapter 15 and the resource factory in Chapter 20.

You can clean up all the resources allocated to the ORB core during initialization by calling CORBA::ORB::destroy(). For example:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// ...use the ORB...

// Release resources.
orb->destroy();
```

### 19.3 Controlling Service Configurator Behavior

TAO uses the *ACE Service Configurator* framework to support dynamic creation and configuration of its components. See 18.3 for more information on how the service configurator is used in TAO.

Table 19-1 lists the options that influence the behavior of the ORB’s service configurator. They duplicate many of the common options available to applications built using the service configurator framework. Client and server applications based on the service configurator framework may either use these options during ORB initialization or use the corresponding one-letter options supplied to ACE_Service_Config::open(). Each option is described in more detail in the referenced section.

**Table 19-1 Service Configurator Control Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBDaemon</td>
<td>19.8.4</td>
<td>Instruct the process to run as a daemon.</td>
</tr>
<tr>
<td>-ORBSvcConfLoggerKey logger_key</td>
<td>19.8.28</td>
<td>Specify where to write ORB logging output.</td>
</tr>
<tr>
<td>-ORBSvcConf config_file_name</td>
<td>19.8.32</td>
<td>Specify an alternate service configurator file name.</td>
</tr>
<tr>
<td>-ORBSvcConfDirective directive</td>
<td>19.8.33</td>
<td>Pass a single directive directly to the service configurator.</td>
</tr>
</tbody>
</table>
19.4 Controlling Debugging Information

Often, during application development and testing, you will want to have fine-grained control over the amount and type of debugging information you receive from the application. You may not always have a debugger available, such as *gdb* or *dbx*, to debug a running application. For example, a debugger may not be available in an embedded environment.

TAO can provide debugging information at several levels of granularity. Table 19-2 lists options that influence the amount and type of debugging information generated by an application. Each option is described in more detail in the referenced section.

**Table 19-2 Debugging Control Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBDebug</td>
<td>19.8.5</td>
<td>Instruct the process to print debug messages from the ACE Service Configurator component.</td>
</tr>
<tr>
<td>-ORBDebugLevel level</td>
<td>19.8.6</td>
<td>Set the maximum tolerance for debugging messages reported from TAO.</td>
</tr>
<tr>
<td>-ORBLogFile file</td>
<td>19.8.18</td>
<td>Redirect all <code>ACE_DEBUG</code> and <code>ACE_ERROR</code> output to a file.</td>
</tr>
<tr>
<td>-ORBObjRefStyle style</td>
<td>19.8.24</td>
<td>Specify the format used to print Interoperable Object References (IORs).</td>
</tr>
<tr>
<td>-ORBVerboseLogging n</td>
<td>19.8.37</td>
<td>Controls the amount of status data printed on each line of the debug log. Higher numbers generate more output.</td>
</tr>
</tbody>
</table>

19.5 Optimizing Request Processing

Often it is possible to achieve better performance and reduce latency by optimizing certain stages of request processing in the ORB. Table 19-3 lists options to control various optimizations during request processing. Each option is described in more detail in the referenced section.

**Table 19-3 Request Processing Optimization Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBCDRTradeoff maxsize</td>
<td>19.8.1</td>
<td>Control the trade-off strategy between copy and no-copy marshaling of octet sequences.</td>
</tr>
</tbody>
</table>
19.6 Connection Management and Protocol Selection

ORBs send and receive requests and replies using various messaging and transport protocols. Each protocol has its own concept of an endpoint. Table 19-4 lists options to manage connections and to control protocol selection within your application. Each option is described in more detail in the referenced section.

Table 19-4 Connection Management and Protocol Selection Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBDefaultInitRef URL_prefix</td>
<td>19.8.7</td>
<td>Specifies default URL prefix to apply when resolving initial object references.</td>
</tr>
</tbody>
</table>
## ORB Initialization Options

### Table 19-4 Connection Management and Protocol Selection Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ORBInitRef ObjectID=IOR</code></td>
<td>19.8.13</td>
<td>Specify an object reference for an initial service.</td>
</tr>
<tr>
<td><code>-ORBEnforcePreferredInterfaces enforce</code></td>
<td>19.8.11</td>
<td>Specifies whether specification for <code>-ORBPreferredInterfaces</code> option needs to be enforced.</td>
</tr>
<tr>
<td><code>-ORBImplRepoServicePort port</code></td>
<td>19.8.12</td>
<td>Specify the <code>port</code> on which the Implementation Repository Service listens for multicast requests.</td>
</tr>
<tr>
<td><code>-ORBLingerTimeout timeout</code></td>
<td>19.8.16</td>
<td>Set the linger timeout on a TCP socket before closing it</td>
</tr>
<tr>
<td><code>-ORBNameServicePort port</code></td>
<td>19.8.20</td>
<td>Specify the <code>port</code> on which the Naming Service listens for multicast requests.</td>
</tr>
<tr>
<td><code>-ORBPreferredInterfaces list</code></td>
<td>19.8.25</td>
<td>Affects how network interfaces are selected on multi-homed hosts.</td>
</tr>
<tr>
<td><code>-ORBTradingServicePort port</code></td>
<td>19.8.34</td>
<td>Specify the <code>port</code> on which the Trading Service listens for multicast requests.</td>
</tr>
<tr>
<td><code>-ORBUseSharedProfile enabled</code></td>
<td>19.8.36</td>
<td>Specify whether or not the ORB should combine multiple endpoints into a single profile.</td>
</tr>
<tr>
<td><code>-ORBEndpoint endpoint(s)</code></td>
<td>19.8.10</td>
<td>Specify that the ORB is to listen for requests on the specified <code>endpoint(s)</code>. Deprecated in favor of <code>-ORBListenEndpoints</code>.</td>
</tr>
<tr>
<td><code>-ORBLaneEndpoint laneid endpoint(s)</code></td>
<td>19.8.14</td>
<td>Provide a collection of endpoints for a specific RTCORBA thread pool lane. Deprecated in favor of <code>-ORBLaneListenEndpoints</code>.</td>
</tr>
<tr>
<td><code>-ORBLaneListenEndpoints laneid endpoint(s)</code></td>
<td>19.8.15</td>
<td>A synonym for <code>-ORBLaneEndpoint</code>.</td>
</tr>
<tr>
<td><code>-ORBMulticastDiscoveryEndpoint endpoint</code></td>
<td>19.8.19</td>
<td>Specify the <code>endpoint</code> on which the Naming Service listens for multicast requests.</td>
</tr>
</tbody>
</table>

For IP addresses under IIOP, use dotted decimal notation rather than host name. Other protocols use a suitable character representation of a numeric address.
19.7 Miscellaneous Options

Table 19-5 lists certain miscellaneous ORB initialization options. Each option is described in more detail in the referenced section.

Table 19-5 Miscellaneous Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBId orb_name</td>
<td></td>
<td>Sets the name of an ORB to orb_name.</td>
</tr>
<tr>
<td>-ORBNoProprietaryActivation</td>
<td>19.8.23</td>
<td>A CORBA 3 option to specify that a server should avoid the use of any proprietary activation framework (e.g., registration with an Implementation Repository) upon start up.</td>
</tr>
<tr>
<td>-ORBServerId server_id</td>
<td>19.8.27</td>
<td>A CORBA 3 option to uniquely identify a server to an Implementation Repository (IMR).</td>
</tr>
<tr>
<td>-ORBUseIMR enabled</td>
<td>19.8.35</td>
<td>Enables the use of the Implementation Repository for persistent POAs.</td>
</tr>
</tbody>
</table>

19.8 Option Descriptions

The remainder of this chapter describes the individual ORB initialization options that may be passed, either programmatically or from the command line, to the ORB initialization function, CORBA::ORB_init().
**ORB Initialization Options**

## 19.8.1 ORBCDRTradeoff `maxsize`

### Values for `maxsize`

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 (default)</td>
<td>The actual default is defined in <code>$ACE_ROOT/ace/OS.h</code> as <code>ACE_DEFAULT_CDR_MEMCPY_TRADEOFF</code>.</td>
</tr>
<tr>
<td>&gt;= 0</td>
<td>The maximum octet sequence size that can utilize the current message block.</td>
</tr>
</tbody>
</table>

### Description

Excessive data copying is a significant source of memory management overhead during request processing. Resizing internal marshaling buffers multiple times when encoding large operation parameters leads to excessive data copying.

TAO minimizes unnecessary data copying by keeping a linked list of Common Data Representation (CDR) buffers. Operation arguments are marshaled into buffers allocated from thread-specific storage (TSS). The buffers are linked together to minimize data copying. Gather-write I/O system calls, such as `::writev()`, can then write these buffers atomically without requiring multiple OS calls, unnecessary data allocation, or copying.

However, if an octet sequence is small and the last buffer in the linked list contains enough unused space for this sequence, copying into that buffer is more efficient than allocating additional buffers and appending them to the linked list.

If the length of the octet sequence is smaller than `maxsize` and there is room in the current message block for it, it will be copied there. This option is useful when applications can predict the octet sequence length and can therefore marshal without copying. By carefully choosing the value of `maxsize`, you can increase marshaling speed, avoid extra pointers and message blocks, and reduce the overall message size.

### Usage

Both server and client applications may use this option. The value of `maxsize` should be chosen to achieve a balance between message block overhead and the copying of octet sequences.

### See Also


### Example

The following example shows how to use the `-ORBCDRTradeoff` command line option to specify a CDR data copy trade-off value of 1024:

```
myserver -ORBCDRTradeoff 1024
```
19.8.2 ORBCollocation is_allowed

Values for is_allowed

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td>Objects within the same address space, even if from different ORBs, are considered collocated.</td>
</tr>
<tr>
<td>yes</td>
<td>Deprecated - means the same thing as global.</td>
</tr>
<tr>
<td>per-orb</td>
<td>Only objects from the same ORB are considered collocated.</td>
</tr>
<tr>
<td>no</td>
<td>Assume collocation is not possible and avoid testing for it.</td>
</tr>
</tbody>
</table>

Description
When a client invokes an operation on a CORBA object whose servant is in the same address space (i.e., the same process), the client and servant are said to be collocated. TAO optimizes collocated client/servant configurations by generating collocation stubs for the client. These stubs bypass several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic, resulting in dramatically increased performance and decreased latency. This option controls in which situations TAO will use the collocation stubs.

Usage
Normally, allowing for collocation is desirable. When the collocation optimization is turned off, every request, even on collocated objects, involves calling through the ORB and into the kernel, at least to the level of a loopback connection. However, if you are certain an application cannot use collocation, you can explicitly disallow the collocation optimization to avoid the slight overhead of determining whether a client and servant are collocated. You may also direct the IDL Compiler to not generate the collocation stubs and thus achieve a smaller memory footprint (see 5.13 for details).

Both server and client applications may use this option.

Note
Collocation may not be desirable for some real-time applications since collocated function invocations are run in the client’s thread-of-control and this can cause priority inversions under some circumstances.

See Also
19.8.3, 19.8.8

Example
The following example shows how to use the -ORBCollocation command line option to disallow the collocation optimization.

```bash
myserver -ORBCollocation no
```

In addition, for an example that utilizes this option, see

```
$TAO_ROOT/performance-tests/Cubit/TAO/IDL_Cubit/collocation_test.cpp
```
ORB Initialization Options

19.8.3 ORBCollocationStrategy \textit{strategy}

\textbf{Values for \textit{strategy}}

\begin{tabular}{ll}
\textit{thru\_poa} (default) & TAO uses a collocated object implementation that respects the POA’s current state and policies. \\
\textit{direct} & Invocations on collocated objects become direct calls to the servant without checking the POA’s status. \\
\end{tabular}

\textbf{Description} As described in 19.8.2, using the collocation optimization allows requests to be dispatched more directly to collocated servants, bypassing several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic. This option controls which of two collocation stubs is used by clients when the collocation optimization is applied.

The default is \textit{thru\_poa}, which will deliver the request through the POA with which the servant is registered. The \textit{direct} strategy will directly deliver the request from the stub to the servant, bypassing the POA. This is a TAO-specific extension and will behave differently in certain situations as described below.

\textbf{Usage} If the \textit{thru\_poa} strategy is used, a \textit{safe} collocated stub is used to handle operation invocations on a collocated object. Though not as fast as a direct virtual function call, these safe collocated stubs are still very efficient, especially compared to normal operation invocations on collocated objects that must go through demarshaling and the loopback interface. Invoking an operation via the safe collocated stub ensures that:

- The servant’s ORB has not been shut down.
- The thread-safety of all ORB and POA operations is maintained.
- The POA that manages the servant still exists.
- The POA Manager of this POA is queried to make sure upcalls are allowed to be performed on the POA’s servants.
- The servant for the collocated object is still active.
- The \texttt{POACurrent}’s context is set up for this upcall.
- The POA’s threading policy is respected.

Using the \textit{direct} strategy optimizes for the common case and ensures that performance is the same as for a direct virtual function call. Invoking an
operation via the direct collocated stub causes the following non-standard behaviors:

- The POACurrent is not set up.
- Interceptors are bypassed.
- POA Manager state is ignored.
- Servant Managers are not consulted.
- Etherealized servants can cause problems.
- Location forwarding is not supported.
- The POA’s Thread_Policy is circumvented.

**Note**  
*Using the direct collocation strategy is not CORBA-compliant. It may also be less reliable than the thru_poa strategy for the reasons cited above. However, direct invocations on collocated objects may be desirable in some real-time applications with very stringent latency requirements.*

**See Also** 19.8.2, 19.8.8

**Example** The following example shows how to use the -ORBColocationStrategy command line option to allow direct operation invocations on collocated objects, thereby bypassing the POA altogether:

```
myserver -ORBColocationStrategy direct
```

In addition, for an example that utilizes this option, see

```
$TAO_ROOT/performance-tests/Cubit/TAO/IDL_Cubit/collocation_test.cpp.
```
ORB Initialization Options

19.8.4 ORBDaemon

**Description** Use `-ORBDaemon` to instruct the process to run as a daemon. In a UNIX environment, this means several things:

- The TTYs (terminals) associated with the process, including `stdin`, `stdout`, and `stderr`, are closed.

- The process will ignore `SIGHUP` signals, so signals generated from the terminal on which the process was launched will not affect it.

- The working directory of the daemon process is set to the root directory, so the file system from which it was launched can be safely unmounted while it is running.

- The `umask` settings of the daemon process are cleared, so the permission bits in the inherited file mode creation mask do not affect the permission bits of new files created by the process.

Using `-ORBDaemon` on the command line is equivalent to passing the `-b` option to the `open()` function of the `ACE_Service_Config` class.

**Usage** Both server and client applications may use this option, but it is most applicable to long-running servers that need to run in the background, disconnected from any terminals. Since it is a command line option, you can still interact with these applications during development and testing, then use the option upon deployment. It uses mechanisms already available through the supplied service configurator, saving you the work of implementing this behavior yourself.

This option is effective only in environments where `ACE::fork()` is implemented. This excludes, for example, Win32 and VxWorks.

**Example** The following example shows how to use the `-ORBDaemon` command line option to daemonize a server process:

```
myserver -ORBDaemon
```
19.8.5 ORBDebug

Description  The -ORBDebug option enables the printing of ACE and TAO debug messages generated by the ACE Service Configurator framework. You can use this option on the command line to enable the printing of additional debug information during development and testing, then disable the printing of such information (by not including the option on the command line) during deployment. Using -ORBDebug on the command line is equivalent to passing the -d option to the open() function of the ACE_Service_Config class.

Usage  Both server and client applications may use this option. It affects the printing of debugging information from the TAO ORB core, connection handling, and protocol handling code.

See Also  18.4, 19.8.6

Example  The following example shows how to use the -ORBDebug command line option to enable the output of debugging information from a server process.

    myserver -ORBDebug
ORB Initialization Options

19.8.6 ORBDebugLevel level

Values for level

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (default)</td>
<td>No optional debugging messages are printed.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Debugging messages with a threshold less than level will be printed.</td>
</tr>
</tbody>
</table>

Description
Sets the maximum tolerance for debugging messages reported from TAO. The default is to not print optional debugging messages.

Usage
Use this option for fine-grained control over the amount of debugging information printed from applications. The environment variable TAO_ORB_DEBUG supersedes this option. Both server and client applications may use this option.

Nearly all messages controlled by the debug level have a threshold of 0. Thus, choose a debug level of 1 to see most debugging messages.

GIOP commands (sent and received) will dump information about messages if the debug level is greater than 4, and will include the message contents if the level is set greater than 9.

GIOP synchronous invocations will report exceptions raised during invocation if the debug level is greater than 5.

See Also
18.4, 19.8.5

Example
The following example shows how to use the -ORBDebugLevel command line option to control the level of debugging information printed from a server.

```
myserver -ORBDebugLevel 6
```
19.8.7 **ORBDefaultInitRef URL_prefix**

**Description** The `-ORBDefaultInitRef` option supplies the ORB with a default URL prefix to be used in resolving object IDs that have not been mapped to a URL by the `-ORBInitRef` option.

When an application calls `resolve_initial_references("ObjectID")`, the ORB processes the call as follows:

If `ObjectID` is found in the `-ORBInitRef` mapping table, the associated URL is immediately resolved into an object reference.

If `ObjectID` is not found in the `-ORBInitRef` mapping table, then:

- If the ORB was supplied with a default URL prefix through the `-ORBDefaultInitRef` option,
  - A URL is constructed by appending a protocol-specific object key delimiter (e.g., ‘/’) to `URL_prefix`, then appending `ObjectID` to the resulting string (e.g. `URL_prefix/ObjectID`).
  - The ObjectURL is then resolved into an object reference.

- If the ORB was not supplied with a default URL prefix, a built-in mechanism (such as IP multicast query) may be used to locate the `ObjectID` object.

**Note** *TAO does not support the OMG-recommended http and ftp formats.*

**Usage** Both server and client applications may use this option. Although it is not an error to repeat this option, each successive use of it will overwrite the previous one, so there is no advantage to be gained from repeating it.

When many initial services share a common URL prefix, it can be convenient to provide a `-ORBDefaultInitRef` as an alternative to a separate `-ORBInitRef` for each service.

The `-ORBDefaultInitRef` option should be used with caution, because it overrides the built-in resolution mechanism for every initial service, except those specified by `-ORBInitRef`. For example, suppose that an application uses both the Naming Service and the Trading Service, and that these services do not share the same address. If the Trading Service is specified through the
ORB Initialization Options

-ORBInitRef option, then resolve_initial_references() will use the built-in mechanism to resolve the Naming Service.

However, if the Trading Service address is specified through -ORBDefaultInitRef, and the Naming Service is not specified by -ORBInitRef, then resolve_initial_references() will use the Trading Service address to construct an ObjectURL for both the Naming Service and the Trading Service. Any attempt by the application to connect to the Naming Service will fail.

The format for URL_prefix is:

<URL_prefix> = <corbaloc> | <protocol>
<corbaloc> = "corbaloc:" [<protocol_id>] ":" <address_list>
<protocol> = [<<protocol_id_loc> "://"] <address_list>
<protocol_id> = "iiop" | "uiop" | "shmiop" | <ppiop>
<protocol_id_loc> = <iiop> | <uiop> | <shmiop> | "mcast" | <ppiop>
<iiop> = "iiop" | "iioploc"
<uiop> = "uiop" | "uioploc"
<shmiop> = "shmiop" | "shmioploc"
<ppiop> = pluggable protocol identifier added to TAO
<address_list> = [<<address>>","]* <address>
<address> = <iiop_addr> | <uiop_addr> | <shmiop_addr> | <mcast_addr> | <ppiop_addr>

<iiop_address> = [<<version> <host> [":" <port>]]
<uiop_addr> = [<<version> <path> ["/" <filename>]]
<shmiop_addr> = [<<version> | <host> [:] <port>]
<mcast_addr> = [<<mcast_group> ":" [<<mcast_port> ":" [<<nic_addr> ":" [<<tll>]]]

Use of the new corbaloc syntax is preferred. The use of all other URLs, such as iioploc, is deprecated.

For corbaloc, the default protocol is “iiop”, and the default port is 2809. corbaloc also supports the “rir” (resolve initial reference) protocol, but this is not applicable to -ORBDefaultInitRef or -ORBInitRef because we are defining the initial references. See 24.4.1 for more details on corbaloc.
Multicast allows clients to discover the address of a service. The service listens on the multicast address for requests from clients and the service responds with the service’s address. The default initial reference is `mcast://:::`.

For `<mcast_addr>`, the default multicast group is 224.9.9.2. This multicast group must be a class D address in the range 224.0.0.0 to 239.255.255.255. The default multicast port is 10013. The default NIC is `eth0`. The default TTL value is 1. TTL is the Time To Live—the number of hops outgoing packets will travel. A value of 1 means outgoing packets will only travel as far as the local subnet.

---

**Note** If `-ORBInitRef` is used to define a particular `ObjectID`, that definition will take precedence over `-ORBDefaultInitRef`.

---

**See Also** 19.8.10, 19.8.13
For more information on using TAO’s pluggable protocols, see Chapter 15.

**Examples** **Using multicast for service discovery**
The following examples show how to use the `mcast:` protocol to find a service.

Start the Naming Service listening to multicast requests by specifying the `-m 1` option:

```
Naming_Service -m 1
```

Start an application server without specifying an initial reference for the Naming Service, thereby relying on the default multicast discovery behavior:

```
myserver
```

The invocation of

```
orb->resolve_initial_references("NameService");
```

1. If a `<object_key>` is appended, as allowed in `-ORBInitRef`, the default is different for the different well-defined services. 10013 is the default if an `<object_key>` is not provided or “NameService” is the `<object_key>`. See `$TAO_ROOT/tao/default_ports.h` for the other defaults.
ORB Initialization Options

from within myserver causes the ORB to retrieve the object reference using multicast.

If you want to specify a different multicast group instead of the default, you can use the following syntax:

```
Naming_Service -ORBMulticastDiscoveryEndpoint 234.5.6.7:8910 -m 1
myserver -ORBDefaultInitRef mcast://234.5.6.7:8910
```

**Using both -ORBInitRef and -ORBDefaultInitRef**
The following example shows how to use both -ORBDefaultInitRef and -ORBInitRef to specify the locations of the Naming Service, Trading Service, and ImplRepo Service.

Start a Naming Service on tango:

```
Naming_Service -ORBListenEndpoints iiop://tango:6666
```

Start another Naming Service on salsa:

```
Naming_Service -ORBListenEndpoints iiop://salsa:7777
```

Start the Trading Service on polka:

```
Trading_Service -ORBListenEndpoints iiop://polka:8888
```

Start the ImplRepo Service on waltz:

```
ImplRepo_Service -ORBListenEndpoints iiop://waltz:2809
```

Start an application server:

```
myserver -ORBInitRef TradingService=corbaloc::polka:8888 \ 
-ORBDefaultInitRef corbaloc::tango:6666,salsa:7777,waltz
```

The invocation of

```
orb->resolve_initial_references("TradingService");
```

from within myserver causes the ORB to attempt `string_to_object()` with the following URL:
corbaloc:iiop:polka:8888/TradingService

When myserver attempts to connect to the Trading Service, the ORB will attempt to connect on polka:8888.

The invocation of

orb->resolve_initial_references("NameService");

from within myserver causes the ORB to attempt string_to_object() with the following URLs:

corbaloc:iiop:tango:6666/NameService
corbaloc:iiop:salsa:7777/NameService
corbaloc:iiop:waltz:2809/NameService

or equivalently:

corbaloc::tango:6666,:salsa:7777,:waltz:2809/NameService

When myserver attempts to connect to the Naming Service, the ORB will attempt to connect first on tango:6666, then on salsa:7777. If both of these attempts fail, it will make an attempt to connect on waltz:2809, but this attempt will fail because the Naming Service is not running on waltz.

The invocation of

orb->resolve_initial_references("ImplRepoService");

from within myserver causes the ORB to attempt string_to_object() with the following URLs:

corbaloc::tango:6666,:salsa:7777,:waltz:2809/ImplRepoService

When myserver attempts to connect to the Implementation Repository, the ORB will attempt to connect first on tango:6666, then on salsa:7777, but these attempts will fail because the ImplRepo Service is not running on either tango or salsa. The ORB will then attempt to connect on waltz:2809, where the ImplRepo Service was started.
**ORB Initialization Options**

**Multiple services share the same address**
The following example shows how to specify a default URL prefix for a Naming Service and Trading Service that share the same address.

Assume that a custom driver called Finder has been written for a Naming Service and Trading Service that share the same ORB. (For an example of a custom Naming Service driver, see 24.6.2.)

Start both the Naming Service and the Trading Service on `tango:9999`

Finder -ORBListenEndpoints iiop://tango:9999

The application server may be started using either the `-ORBInitRef` option

```
myserver -ORBInitRef NameService=corbaloc::tango:9999/NameService -ORBInitRef TradingService=corbaloc::tango:9999/TradingService
```

or the `-ORBDefaultInitRef` option (much less typing involved)

```
myserver -ORBDefaultInitRef corbaloc::tango:9999
```

In either case, the invocation of

```
orb->resolve_initial_references("NameService");
```

from within `myserver` causes the ORB to use the following URL:

`corbaloc:iiop:tango:9999/NameService`

The invocation of

```
orb->resolve_initial_references("TradingService");
```

causes the ORB to use the following URL:

`corbaloc:iiop:tango:9999/TradingService`

Thus, when `myserver` attempts to connect to either the Naming Service or the Trading Service, the ORB will attempt to connect on `tango:9999`.

**Some Valid Examples**

```
-ORBDefaultInitRef corbaloc::tango://uses IIOP and port 2809
-ORBDefaultInitRef corbaloc:iiop:tango:9999
```
-ORBDefaultInitRef corbaloc::tango::salsa::8888,iiop::tango::9999

**Some Invalid Examples**

-ORBDefaultInitRef host:port // not corbaloc
-ORBDefaultInitRef corbaloc:iiop:port // no host
19.8.8 ORBDisableRTCollocation boolean

Values for boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (default)</td>
<td>The default value leaves real-time collocation resolution decisions to the real-time collocation resolver used by the RT CORBA ORB.</td>
</tr>
<tr>
<td>1</td>
<td>Do not use the real-time collocation resolver; instead, rely on the ORB’s default collocation resolution method.</td>
</tr>
</tbody>
</table>

Description

This option controls how collocation optimization decisions are made in RT CORBA applications. As described in 19.8.2, TAO normally optimizes collocated invocations (where the client and the target object are in the same address space). The effect of the ORB’s default collocation optimization is such that the client thread is used to carry out the request. As described in 17.4.5, this effect may be undesirable in real-time applications. Therefore, TAO’s implementation of RT CORBA employs a special “real-time collocation resolver” (RT_Collocation_Resolver) to determine whether an invocation should be subject to collocation optimization. The RT_Collocation_Resolver considers the following factors when making collocation decisions:

- The ORB and POA of the target object.
- The thread pool policy of the target’s POA.
- The thread pool id and thread pool lanes of the target’s POA.
- The priority model of the target’s POA.
- The invoking thread.

These factors are considered in making collocation decisions to ensure the request is carried out at the appropriate priority.

However, not all applications need such a precise definition of collocation. For these applications, the -ORBDisableRTCollocation option can be used to bypass the real-time collocation resolver and use the ORB’s default collocation resolution method, as described in 19.8.2 and 19.8.3. A value of 1 (true) disables real-time collocation resolution decisions and falls back on the default collocation decisions implemented in the default ORB. This behavior may result in better performance, but the invocation may not be subject to the appropriate RT CORBA thread and priority constraints.

The default value of this option is 0.
**Usage**  
Both server and client applications may use this option. It only affects client invocations on objects within the same address space.

**See Also**  
19.8.2, 19.8.3, Chapter 9

**Example**  
The following example shows how to use the `-ORBDisableRTCollocation` option to specify that the default ORB’s collocation resolution method should be used instead of the real-time collocation resolver.

```
myserver -ORBDisableRTCollocation 1
```
ORB Initialization Options

19.8.9 ORBDottedDecimalAddresses enabled

Values for enabled

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (default)</td>
<td>Use host/domain names in IORs.</td>
</tr>
<tr>
<td>1</td>
<td>Use IP addresses in IORs.</td>
</tr>
</tbody>
</table>

Description
An IOR exported by an ORB contains information about the communication endpoint of the server. When TCP/IP is used for communication between ORBs, the server endpoint is identified by a host-and-port tuple of the form host:port. The host name is typically specified as a domain name (e.g., tango.acme.com) rather than an IP address (e.g., 128.252.165.61). This allows Domain Name Servers (DNS) and routers to make intelligent choices about what network routes clients will use in communicating with the server. It also allows network managers to facilitate rudimentary load balancing by reconfiguring a DNS to return different IP addresses for a particular domain name. However, there are occasions when the routers and DNS’s of a network do not know about a particular domain name. In this case, the only way for the client to connect to the server is to use the IP address, bypassing the domain-name-to-IP-address conversion.

TAO will encode any endpoints specified through the -ORBEndpoint or -ORBListenEndpoints options or the TAO_ORBENDPOINT environment variable exactly as they are given. The -ORBDottedDecimalAddresses option will override this behavior.

When using an IP address, there is a slight decrease in latency during the initial connection between client and server ORBs since no DNS look-up is performed.

Usage
Both server and client applications may use this option. Provide a value of 1 to instruct the ORB to encode IP addresses, instead of domain names, in IORs.

Note
Since dotted decimal address specifications apply only to IP networks, and not all ORB protocols are implemented on IP, this option is currently treated as a suggestion for each loaded protocol to use a character representation for the numeric address (if enabled=1), otherwise to use a logical name.
Examples

The following examples show how to use the `-ORBDecimalAddresses` option to specify that object references generated by the ORB should have IP addresses instead of host/domain names encoded within them:

```
myserver -ORBDecimalAddresses 1
myserver -ORBDecimalAddresses 1 -ORBListenEndpoints iiop://host.domain.com
```
19.8.10 **ORBEndpoint endpoint(s)**

**Note**  
The TAO-specific `-ORBEndpoint` option has been deprecated in favor of the OMG-standard `-ORBListenEndpoints` option. See 19.8.17.

**Description**  
Tells the ORB to listen for requests on the interfaces specified by `endpoint(s)`. Endpoints are specified using a URL. An endpoint has the form:

```
protocol://V.v@addr1,...,W.w@addrN
```

where `V.v` and `W.w` are optional protocol versions for each address. An example of an IIOP endpoint is:

```
iiop://hostname:port
```

Sets of endpoints may be specified using multiple `-ORBEndpoint` options or by delimiting endpoints with a semi-colon (`;`). For example,

```
-ORBEndpoint iiop://localhost:9999 -ORBEndpoint uiop:///tmp/mylocalsock
-ORBEndpoint shmiop://10002
```

is equivalent to

```
-ORBEndpoint 'iiop://localhost:9999;uiop:///tmp/mylocalsock;shmiop://10002'
```

Note the single quotes (`'`) in the latter option specification. Single quotes are needed to prevent the shell from interpreting text after the semi-colon as another command.

If an endpoint is specified without an address, such as

```
-ORBEndpoint uiop:// -ORBEndpoint shmiop://
```

then a default endpoint will be created for the specified protocol.

Valid endpoint protocols include IIOP, UIOP (on platforms that support local IPC), SHMIOP, DIOP, and any additional protocols whose factories were loaded via the resource factory’s `-ORBProtocolFactory` option. See 20.2.6
for more information on protocol factories. See Chapter 15 for information on using TAO’s pluggable protocols.

If the -ORBEndpoint option is not used, then one endpoint is created by each protocol factory that is registered with the resource factory. Typically, this means an IIOP endpoint is provided, using the host name of the local machine and a randomly-selected port number, as if -ORBEndpoint iiop:// had been specified. The value of the port number may be different each time the server is run.

Each endpoint that appears as an argument to the -ORBEndpoint option can itself accept endpoint-specific options. Such options will only apply to the endpoint for which they were specified.

Currently, TAO supports the portspan, ssl_port and hostname_in_iors options. The portspan option allows you to specify a range of ports for the ORB to use for endpoints. The ORB will pick the first port in the range that is not already in use. The portspan option is useful when you want to start several instances of a server on a range of ports or if you want to restrict the range of ports that your servers can use. Use the syntax portspan=value where value is the number of ports in the range. The ssl_port option is used with the TAO SSLIOP pluggable protocol. See 15.11 for more information on using SSLIOP. The hostname_in_iors option is used to explicitly specify the host name used in IORs. The server ORB does not validate the specified host name.

Note You can only use the -ORBEndpoint option with servers. It has no effect on pure clients.

Note The CORBA 3 specification defines a standard ORB initialization option for specifying the endpoints on which the ORB should listen for requests (-ORBListenEndpoints). TAO supports the new -ORBListenEndpoints option and accepts exactly the same endpoint formats for it as it does for the -ORBEndpoint option.
**ORB Initialization Options**

**Note**  
You can also use the TAO ORBENDPOINT environment variable to specify the ORB's listen endpoints. The value of this environment variable has exactly the same syntax as the -ORBEndpoint option. If the TAO ORBENDPOINT environment variable is used in addition to the -ORBEndpoint or -ORBListenEndpoints option, the endpoint(s) specified by the environment variable are added to the list of the endpoints specified by the above options.

**IIOP Endpoints**

TAO’s IIOP pluggable protocol utilizes TCP/IP as its underlying transport mechanism. IIOP endpoints in TAO have the form:

```
-ORBEndpoint iiop://V.v@hostname1:port1,...,W.w@hostname2:port2
```

where “V.v” and “W.w” are the IIOP protocol versions associated with the given address (hostname:port). Currently supported versions are 1.0, 1.1, and 1.2.

Options are separated from addresses by a forward slash (‘/’). For instance, if an IIOP endpoint is to occupy a port in the range from 5000-5009, the following endpoint specification could be used:

```
myserver -ORBEndpoint iiop://host:5000/portspan=10
```

The preceding example will start a server on each of the ports 5000-5009 if you issue the command 10 times. The 11th time will fail, since all ports in the range are already in use.

IIOP addresses are comprised of a hostname (or an IP address) and a TCP port on which the server listens. The hostname is used to select the network interface on which to set up the endpoint and is used to generate the IOR. Suppose a host has the following network interfaces:

```
eth0: foo1.bar.baz (DEFAULT)  
eth1: foo2.bar.baz
```

To set up an endpoint on the second network interface, “eth1,” either of the following endpoint specifications could be used:

```
-ORBEndpoint iiop://foo2
```

or
An available port will be chosen by TAO (actually the operating system kernel) and placed into the IOR.

To set up an endpoint on a specific port, simply use an endpoint of the form:

-ORBEndpoint iiop://foo2:1234

where 1234 is the TCP port on which the endpoint will be opened. In this case, an endpoint will be opened on the network interface associated with the hostname foo2 on port 1234.

Port names are also accepted. For example, if a UNIX installation has a service called "my_protocol" associated with port 1234 in the service database in /etc/services, then the following would cause an endpoint to be opened on the port associated with that service:

-ORBEndpoint iiop://foo2:my_protocol

Port numbers range from 0 (port is chosen by the operating system) to 65535. Port numbers less than 1024 on UNIX systems are considered privileged and require super-user privileges to access them. Also be aware that some ports may already be in use by other applications.

If no address is specified (e.g., -ORBEndpoint iiop://), then an endpoint with an automatically-chosen port number will be set up on each network interface detected by TAO. Each endpoint will use the same port number and will be represented in the generated IOR as a separate profile or as an alternate address within a single IOR profile. Note that network interface detection will only work on platforms that support this feature. If network interface detection is not supported, then the default network interface will be chosen.

Using a specification of the form -ORBEndpoint iiop://:1234 will create an endpoint with TCP port 1234 on each detected network interface. Note that there is a colon (":") preceding the port number 1234. That colon is necessary for TAO to interpret 1234 as a port. Without the colon, TAO would interpret 1234 as a hostname associated with a given network interface.

Note  Testing under Windows XP/2000 has shown that the gethostname() function, used internally, may not return the fully-qualified domain name (i.e.,
ORB Initialization Options

returning “host” instead of “host.domain.com”). You may want to use the
-ORBEndpoint option on Windows XP/2000 to ensure that your servers
generate IORs that are usable from outside the local network.

SHMIOP Endpoints

TAO’s SHMIOP pluggable protocol utilizes shared memory as its underlying
transport mechanism. SHMIOP endpoints in TAO have the form

-ORBEndpoint shmiop://V.v@port1,...,W.w@port2

where “V.v” and “W.w” are the SHMIOP protocol versions associated with
the given address (port). Currently supported versions are 1.0, 1.1, and 1.2.

SHMIOP addresses consist of a port number on which the server listens. Port
numbers range from 0 (port is chosen by operating system) to 65535. Port
numbers less than 1024 on UNIX systems are considered privileged and
require super-user privileges to access them. Also be aware that some ports
may already be in use by other applications.

TAO will automatically choose an address for a SHMIOP endpoint if the
address is omitted from the specification (i.e., -ORBEndpoint shmiop://).

UIOP Endpoints

TAO’s UIOP pluggable protocol utilizes local IPC (i.e., UNIX domain
sockets) as its underlying transport mechanism.

UIOP endpoints in TAO have the form:

-ORBEndpoint uiop://V.v@rendezvous_point1,...,W.w@rendezvous_point2

where “V.v” and “W.w” are the UIOP protocol versions associated with the
given rendezvous point. Currently supported versions are 1.0, 1.1, and 1.2.

A UIOP address is the rendezvous point on which the server listens. This
rendezvous point is generally the full path to the desired UNIX domain socket
filename. Though relative paths can be used, their use is discouraged. The
maximum length of the rendezvous point is 108 characters, as dictated by the
POSIX.1g specification for local IPC rendezvous points. TAO will truncate
any rendezvous point name longer than 108 characters.
A UIOP endpoint with the \textit{absolute} path rendezvous point /tmp/foobar is created by specifying -ORBEndpoint uiop:///tmp/foobar, where the optional protocol version and endpoint-specific options have been omitted.

A UIOP endpoint with the \textit{relative} path rendezvous point foobar is created in the current directory by specifying -ORBEndpoint uiop://foobar, but rendezvous points with relative paths are discouraged because it is possible that other rendezvous points with the same base name exist on a given system, giving rise to potential ambiguities.

Omitting the rendezvous point (i.e., specifying -ORBEndpoint uiop://) will cause TAO to automatically create an absolute-path rendezvous point. The rendezvous point will be located in a system temporary directory and its name will begin with “TAO.”

\textbf{SSLIOP Endpoints}
SSLIOP stands for the Secure Sockets Layer (SSL) Inter-ORB Protocol. This protocol is defined by the OMG as part of the CORBA Security Service specification. SSLIOP uses GIOP as a messaging protocol and SSL as the transport protocol. It is a drop-in replacement for IIOP, providing secure communication between hosts.

SSLIOP endpoints are specified similarly to IIOP endpoints. An SSLIOP endpoint is specified just like an IIOP endpoint with the addition of the \texttt{ssl\_port} endpoint-specific option:

\begin{verbatim}
-ORBEndpoint iiop://hostname:iiop\_port/ssl\_port=secure\_port
\end{verbatim}

where the hostname and IIOP port are defined just like in IIOP endpoints and the \texttt{ssl\_port} option specifies the port that will be used to establish a secure connection for secure communications.

SSLIOP is described in more detail in 15.11 and 29.9.

\textbf{DIOP Endpoints}
DIOP stands for Datagram Inter-ORB Protocol and is a UDP-based transport protocol. This protocol is only partially implemented; as such, there are restrictions on its use. The DIOP implementation uses connectionless UDP sockets, and therefore is intended for use as a low-overhead protocol for certain classes of applications. The original motivation for this protocol was applications that use only \texttt{oneway} operations.
Endpoints for DIOP are composed of the prefix “diop://”, followed by a host and port combination, similar to IIOP endpoints. An example of a DIOP endpoint is:

-ORBEndpoint diop://example.ociweb.com:12345

DIOP is described in more detail in 15.10.

**Endpoint-specific Options**

An endpoint-specific option is used as follows:

-ORBEndpoint iiop://foo:1234/option=value

Additional options can be specified by separating each option with an ampersand (‘&’) as follows:

-ORBEndpoint 'iiop://foo:1234/option1=value1&option2=value2'

Note that the address and the endpoint-specific options are separated by a forward slash (‘/’) in this case, i.e., for IIOP endpoints. This character is normally a slash (‘/’), but may differ for other types of pluggable protocol endpoints. For example, UIOP endpoint-specific options are separated from the address by a vertical bar (‘|’). Also note that when using more than one option, quotes should be used to prevent the shell from interpreting the ampersand (‘&’) as indicating that the process should be run in the background.

**See Also**


**Examples**

**IIOP Endpoint Examples**

In the following example, we use the -ORBEndpoint command line option to restrict IIOP connections to clients running on the same host as the server by specifying localhost as the host name of the endpoint. No port number is specified, so it will be selected randomly. The command line is:

```
myserver -ORBEndpoint iiop://localhost:
```

Here are some additional examples of IIOP endpoints:

-ORBEndpoint iiop://1.1@foo:0
-ORBEndpoint iiop://1.1@foo:0,1.2@bar,baz:3456
-ORBEndpoint iiop://1.1@foo:0,1.2@bar,baz:3456/portspan=10
-ORBEndpoint iiop://portspan=5 (note three slashes "///")
-ORBEndpoint iiop://2020/portspan=20

**SHMIOP Endpoint Examples**
Here are some additional examples of SHMIOP endpoints:

-ORBEndpoint shmiop://1.1@0
-ORBEndpoint shmiop://1.1@0,3456

**UIOP Endpoint Examples**
Here are some additional examples of UIOP endpoints:

-ORBEndpoint uiop://1.1@/tmp/foo1
-ORBEndpoint uiop://1.1@/tmp/foo,1.2@/home/bar/baz

**Multiple Protocol Endpoint Examples**
The following example shows how to use the -ORBEndpoint command line option to specify that the ORB is to listen on an IIOP endpoint at port number 12345 on host tango and at the same time on a default UIOP endpoint.

myserver -ORBEndpoint iiop://tango:12345 -ORBEndpoint uiop://

Here are some additional examples of multiple protocol endpoint specification:

-ORBEndpoint 'iiop://1.1@foo1:0;shmiop://1.1@0,3456;uiop://1.1@/tmp/foo1'
19.8.11 ORBEnforcePreferredInterfaces enforce

Values for enforce

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>Enforce use of preferred interfaces only, thereby causing the ORB to raise an exception if preferred interfaces cannot be used to connect to a target host/interface.</td>
</tr>
<tr>
<td>no (default)</td>
<td>Do not enforce use of preferred interfaces only.</td>
</tr>
</tbody>
</table>

Description
This option specifies whether the -ORBPreferredInterfaces option is enforced. This option can help in cases where the preferred local network may not have a route to the target host/interface. The ORB can choose a default local interface to send the message, or it can raise an exception to the application that preferred interfaces are not usable. This option determines the ORB’s behavior in this case. If the value of the -ORBEnforcePreferredInterfaces option is set to yes, unusable preferred interfaces will make the ORB raise an exception to the application. The default value for this option is no.

Note
Not all transport protocols are based on TCP/IP. Thus, the arguments provided to such options may be interpreted in a protocol-specific way.

Usage
This option is only significant for ORBs in a client role and when the -ORBPreferredInterfaces option is also used.

Example
The following example shows how to use the -ORBEnforcePreferredInterfaces command line option:

```bash
myclient -ORBPreferredInterfaces "*.sometargethost.com:malory.ociweb.com" -ORBEnforcePreferredInterfaces yes
```
19.8.12 ORBImplRepoServicePort \textit{port}

\textbf{Description} \hspace{1em} When \texttt{CORBA::ORB::resolve_initial_references("ImplRepoService")} is called, TAO by default uses IP multicast to find the TAO Implementation Repository (ImplRepo) Service. Often, more than one ImplRepo service is running on a local network during development or in a deployed application where a large number of objects are logically partitioned in some way. In such cases, it is undesirable for all the ImplRepo service servers to be listening for multicast requests on the same port.

At startup, the TAO ImplRepo service is assigned a port on which to listen for multicast requests. The \texttt{-ORBImplRepoServicePort} option allows the user to specify this port. If this option is not used, the \texttt{ImplRepoServicePort} environment variable (if set) is used. If this option is not used and the environment variable is not set, then the port number is set to the value of \texttt{TAO_DEFAULT_IMPLREPO_SERVER_REQUEST_PORT}, defined in \texttt{$TAO\_ROOT/tao/default_ports.h} as \texttt{10018}.

\textbf{Usage} \hspace{1em} Use \texttt{-ORBImplRepoServicePort} when starting the TAO ImplRepo Service to specify on which port it is to listen for multicast requests. Use it when starting clients of the ImplRepo Service to tell them the multicast port to use when multicasting a TAO ImplRepo service discovery request. This option is only available on platforms that support IP multicast. Both server and client applications may use this option.

This option has the following format:

\texttt{-ORBImplRepoServicePort port}

where \texttt{port} is the ImplRepo service’s multicast request port number. This port number must be valid for your operating system and user permissions.

\textbf{Example} \hspace{1em} The following example shows how to use the \texttt{-ORBImplRepoServicePort} command line option to specify the multicast request port to use.

Start ImplRepo service and tell it to listen on port 1234 for multicast requests:

\texttt{$TAO\_ROOT/orbsvcs/ImplRepo_Service/ImplRepo_Service -ORBImplRepoServicePort 1234}

Start an application and tell it to use port 1234 for sending ImplRepo service multicast requests:

\texttt{myserver -ORBImplRepoServicePort 1234}
ORB Initialization Options

19.8.13 ORBInitRef ObjectID=ObjectURL

Description Using the -ORBInitRef option causes the ORB to create a mapping between a specified object identifier (ObjectID) and a specified object reference (ObjectURL). The ORB maintains a table of these mappings for use in the processing of calls to CORBA::ORB::resolve_initial_references().

When an application calls resolve_initial_references("Obj_ID"), the ORB processes the call as follows:

- If Obj_ID is found in the mapping table, the associated ObjectURL is immediately resolved into an object reference.
- If Obj_ID is not found in the mapping table, then
  - If the ORB was supplied with a default URL prefix through the -ORBDefaultInitRef option, an object URL is constructed from this prefix, then resolved into an object reference (see 19.8.7).
  - If a default URL prefix was not supplied, a built-in mechanism (such as IP multicast query) may be used to locate the Obj_ID object.

ObjectID represents an object identifier that may be defined either by TAO or by the application developer. TAO reserves the following ObjectID names: NameService, TradingService, ImplRepoService.²

ObjectURL is a “stringified” object reference that satisfies one of the following URL formats (see 19.8.10):

- IOR: format, e.g.,
  IOR:000000000000000001749444c3a4d4432f436f6e74726f6c65723a...

- protocol_id: format, e.g.,
  corbaloc::tango:9999/TradingService

- file: format, e.g.,
  file:///home/testing/file1.ior

Each of these formats directly specifies an address list and an object key.

Usage Both server and client applications may use this option. It can be used either to configure the ORB at run time with new initial service ObjectIDs that were not defined when the ORB was installed or to override default initial service

² The names IORManipulation, ORBPolicyManager, POACurrent, PolicyCurrent, RootPOA, and TypeCodeFactory are also reserved, but are never used with -ORBInitRef.
resolution mechanisms established for standard services (e.g., the Naming Service or Trading Service) at installation. It takes precedence over the use of the prefix passed with -ORBDefaultInitRef. This option may be repeated.

The ObjectURL formats are described below:

- The IOR: format is simply a “stringified” IOR that consists of the string “IOR:” plus a sequence of hexadecimal digits that encode the address list, object key, and possibly other information, e.g.,

  -ORBInitRef NameService=IOR:000000000001749444c3a4d44432f436f6e7...

  The stringified IOR is normally exported by the object to be referenced, then captured somehow by the client. As such, it is most often used with the file://path/filename format described below.

- The protocol_id: format is defined as follows:

\[
\text{<objectURL>} = (\text{<corbaloc>} | \text{<protocol> }) [\text{<delimiter>} \text{<object_key>} ]
\]

\[
\text{<corbaloc>} = "corbaloc:" [\text{<protocol_id>}] ":" \text{<address_list>}
\]

\[
\text{<protocol_id>} = "\text{iiop}" | "\text{uiop}" | "\text{shmiop}" | "\text{ppiop}" 
\]

\[
\text{<address_list>} = \text{< address_list> } 
\]

\[
\text{<address>} = \text{<iiop_prot_addr>} | \text{<uiop_addr>} | \text{<shmiop_addr>} | \text{<mcast_addr>} | \text{<ppiop_addr>}
\]

\[
\text{<iiop_prot_addr>} = \text{<iiop_id>}\text{<iiop_addr>}
\]

\[
\text{<iiop_id>} = ":" | \text{<iiop_prot_token>}
\]

\[
\text{<iiop_addr>} = [\text{<version>} <\text{host}> [":" <\text{port}>]]
\]

\[
\text{<uiop_addr>} = [\text{<version>} <\text{path}> [":" <\text{filename}>]]
\]

\[
\text{<shmiop_addr>} = [\text{<version>} [\text{<host> ":" <host> ["" <filename>]]
\]

\[
\text{<mcast_addr>} = [\text{<version>} [\text{<host> ":" <host> ["" <filename>]]
\]

\[
\text{<ppiop_addr>} = \text{defined by the pluggable protocol}
\]

\[
\text{<host>} = \text{DNS-style_Host_Name | ip_address}
\]

\[
\text{<port>} = \text{number}
\]

\[
\text{<version>} = \text{<major> "." <minor> "@" | empty_string}
\]

\[
\text{<major>} = \text{<minor>} = \text{number}
\]

\[
\text{<path>} = ["/" <directory_name>]*
\]

\[
\text{<filename>} = <directory_name> = string
\]

\[
\text{<delimiter>} = \text{<iiop_del>} | \text{<uiop_del>} | \text{<shmiop_del>} | \text{<mcast_del>} | \text{<ppiop_del>}
\]

\[
\text{<iiop_del>} = \text{<iiop_del>}
\]

\[
\text{<uiop_del>} = \text{<uiop_del>}
\]

\[
\text{<shmiop_del>} = \text{<shmiop_del>}
\]

\[
\text{<mcast_del>} = \"/"
\]

\[
\text{<ppiop_del>} = \"/"
\]
ORB Initialization Options

<ppiop_del> = defined by the pluggable protocol
<object_key> = ObjectID

Use of the new corbaloc syntax is preferred. The use of all other object URLs, such as iioploc, is deprecated.

See 19.8.7 for a description of the mcast: and corbaloc: protocols.

A uiop ObjectURL must be enclosed by single quotes so that the shell does not interpret the vertical bar (‘|’) delimiter as a command (i.e., the “pipe” symbol).

The protocol_id: format is preferred over the IOR and file notation because it provides a stringified object reference that is easily manipulated in TCP/IP- and DNS-centric environments such as the Internet.

- The file://path/filename format is used as follows:
  - Create a URL using either the IOR: or protocol_id: format.
  - Store this URL in a file.
  - Type -ORBInitRef ObjectID=file://path/filename

    where filename is the name of the file in which the URL is stored and path is the full path to the file. The contents of the file are interpreted by -ORBInitRef as a single object reference.

Note TAO does not currently support the OMG-recommended http and ftp formats.

See Also 19.8.7, 19.8.10
For more information on using TAO’s pluggable protocols, see Chapter 15.

Examples The following examples show various ways to use -ORBInitRef.

  protocol_id: format - preferred corbaloc style
  myserver -ORBInitRef NameService=corbaloc::1.1@tango:9999/NameService
  client1 -ORBInitRef NameService=corbaloc:shmiop:2020/NameService

  protocol_id: format - with multiple addresses
  myclient -ORBInitRef \ NameService=corbaloc::tango:9999,:waltz:2809/NameService
When myclient first attempts to connect to the Naming Service, the ORB will attempt to connect on tango:9999. If that fails, it will attempt to connect on waltz:2809.

**IOR: format**

```shell
myserver -ORBInitRef NameService=IOR:0000000000001749444c3a4d44432f43...
```

**protocol_id: format - mcast style**

```shell
myserver -ORBInitRef \
TradingService=mcast://234.1.2.3:12345:eth1:2/TradingService
```

```shell
myclient -ORBInitRef NameService=mcast://:12345::/NameService
```

**protocol_id: format - deprecated iiop | shmiop | uiop style**

```shell
myserver -ORBInitRef TradingService=iiop://traderhost:16001/TradingService
```

```shell
myclient -ORBInitRef NameService=iiop://1.2@tango:9999/NameService
```

**protocol_id: format - deprecated iioploc | shmioploc | uioploc style**

```shell
myserver -ORBInitRef NameService=iioploc://1.1@tango:9999/NameService
```

```shell
client1 -ORBInitRef NameService=shmioploc://2020/NameService
```

Note that the uioploc URL in the next example is enclosed in quotes:

```shell
client2 -ORBInitRef 'NameService=uioploc://1.2@/tmp/foo1|NameService'
```

**Pluggable-protocol-style iop:format example**

```shell
myserver -ORBInitRef NameService=myioploc://1.2@tango:9999/NameService
```

**file: format**

```shell
client1 -ORBInitRef TestService=file:///usr/local/testing/TestService.ior
```
ORB Initialization Options

19.8.14 ORBLaneEndpoint lane endpoint(s)

Description
This option is used for supplying endpoint definitions for specific RTCORBA thread pool lanes. A thread pool is a collection of threads that allow the ORB to process many requests simultaneously. Within a thread pool, groups of threads may be partitioned by runtime priority. These thread partitions are known as lanes. For instance, you might create a process that has separate high priority and low priority thread lanes. Each of these thread lanes must have its own endpoint to ensure that requests of a certain priority are handed by the appropriate threads and there is no chance for priority inversion.

For this endpoint, a “lane” is identified by a string consisting of a thread pool identifier and a lane identifier within that pool. Symbolically, this is represented as <pool_id>:<lane_id>, where pool_id and lane_id are both integers. Unfortunately, the RT CORBA specification contains no information regarding identifiers for pools and lanes, so this identification is dependent upon consistent initialization of the RT ORB and consistent initialization of thread pools. When creating a thread pool with lanes, each lane is defined by a structure, and many lane definitions are combined in a sequence. For a given pool, the lane identifier is the index of the lane’s position in the sequence. The pool identifier is defined by a counter that is an ORB internal resource. This counter starts at 1, and is incremented for each thread pool that is created, whether or not that pool contains thread lanes.

This presents a discontinuity; if you wish to serve persistent objects from a server that uses thread pools with lanes, you must ensure that the thread pools are defined consistently, along with the lane endpoint definitions. The most reliable way to manage this is to generate the -ORBLaneEndpoint arguments programatically, rather than relying on an external script or user to enter them on the command line. An example of this technique is shown in 19.2.

Each lane endpoint definition may include many endpoints, in the same manner as the -ORBEndpoint and -ORBListenEndpoints options.

See Also
19.8.10, 19.8.15, Section 4.5.1.2 of the CORBA 3.0.3 specification (OMG Document formal/04-03-12)

Example
In this example, we configure a single thread pool with two lanes:

```bash
-ORBLaneEndpoint 1:0 iiop://:1234 \\
-ORBLaneEndpoint 1:1 iiop://:1235
```
19.8.15 ORBLaneListenEndpoints *lane endpoint(s)*

Description: Identical to the –ORBLaneEndpoint option. It is presented as an alias to maintain consistency with the TAO specific legacy option, –ORBEndpoint and its new standard alias, –ORBListenEndpoints.

See Also: 19.8.10, 19.8.14, Section 4.5.1.2 of the CORBA 3.0.3 specification (OMG Document formal/04-03-12)
19.8.16 ORBLingerTimeout *timeout*

**Description**  
This option causes the ORB to set the `SO_LINGER` option on TCP sockets, with a specified timeout value in seconds, before closing the sockets. This option is only useful when using IIOP. The timeout value can be in the range of zero to the maximum signed integer value for the particular platform on which TAO is running.

**Usage**  
When a TAO client, configured to use IIOP, encounters an error while writing to a socket, the ORB attempts to close the socket and open a new one. TAO does not use the `SO_LINGER` socket option when sockets are opened. However, on some platforms, the ORB can have problems closing a socket after an error, for example if the other end is not responding, some data has already been written to the socket, and the socket buffers are not empty. If this occurs, the socket may be left in a `FIN_WAIT_1` state. Meanwhile, the ORB may keep opening new sockets to try to continue sending. Eventually, the client may run out of resources (e.g., file descriptors or mbufs).

If the `-ORBLingerTimeout` option is specified, the ORB will set the `SO_LINGER` socket option with the specified timeout value before closing the socket. By specifying a very short timeout value, the ORB can successfully close the socket without waiting for its buffers to become empty.

Since this option uses TCP socket options, it is only useful when using IIOP. The most common use case is to set the timeout value to zero, so sockets are closed immediately when there is an error, thereby helping to avoid resource exhaustion on some platforms.

**Example**  
The following example shows how to use the `-ORBLingerTimeout` option to set the `SO_LINGER` timeout value to zero.

```
myclient -ORBLingerTimeout 0
```
19.8.17 ORBListenEndpoints endpoint(s)

Description
As of the CORBA 3.0 specification (OMG Document formal/02-06-33), there is a standard ORB initialization option for specifying the endpoints on which the ORB will listen for requests. The format of the endpoints argument is left up to the ORB implementation. In TAO, the -ORBListenEndpoints option accepts the same endpoint specifiers and has exactly the same effect as the -ORBEndpoint option. You can simply replace “-ORBEndpoint” with “-ORBListenEndpoints”. Therefore, no further description of -ORBListenEndpoints is provided here. See the description of the -ORBEndpoint option in 19.8.10.

See Also
19.8.10, Section 4.5.1.2 of the CORBA 3.0.3 specification (OMG Document formal/04-03-12)
19.8.18 ORBLogFile file

**Description**
The `-ORBLogFile` option causes all `ACE_DEBUG` and `ACE_ERROR` output to be redirected to `file`.

**Usage**
The default destination for output from `ACE_DEBUG` and `ACE_ERROR` is `stderr`. Use this option to redirect this output to `file`.

This option may be used with the `-ORBDaemon` option (19.8.4) to capture the `ACE_DEBUG` and `ACE_ERROR` output that would otherwise be lost because the `-ORBDaemon` option causes both `stdout` and `stderr` to be closed.

**See Also**
19.8.4, 19.8.5, 19.8.6

**Example**
The following example shows how to redirect `ACE_DEBUG` and `ACE_ERROR` output to the file `myLogFile`:

```
myserver -ORBLogFile /tmp/myLogFile
```

The following example shows how to capture the `ACE_DEBUG` and `ACE_ERROR` output from a daemon process:

```
myclient -ORBDaemon -ORBLogFile /tmp/myLogFile
```
19.8.19 ORBMulticastDiscoveryEndpoint *endpoint*

**Description**  
When `CORBA::ORB::resolve_initial_references("NameService")` is called, TAO, by default, uses IP multicast to find the TAO Naming Service. It is often the case that more than one Naming Service is running on a local network during development, or in a deployed application where a large number of objects are logically partitioned in some way. In such a case, it is undesirable for all of the Naming Service servers to be listening for multicast requests on the same endpoint.

At startup, the TAO Naming Service joins a multicast group and is assigned a multicast request port. The multicast group IP address and the request port together form the endpoint on which the Naming Service listens for multicast requests. The TAO Naming Service default multicast group IP address is defined in `$ACE_ROOT/ace/OS.h` as `224.9.9.2` and the default multicast request port number is defined in `$TAO_ROOT/tao/default_ports.h` as `10013`.

The `-ORBMulticastDiscoveryEndpoint endpoint` option allows the user to specify the multicast group IP address and the multicast request port for the TAO Naming Service. The option `-ORBNameServicePort port` allows the user to set the multicast request port, but does not allow the user to set the multicast group. In fact, `-ORBMulticastDiscoveryEndpoint 224.9.9.2:port` is functionally equivalent to `-ORBNameServicePort port`. Both options can be used to run more than one Naming Service on the same subnet.

**Usage**  
Use `-ORBMulticastDiscoveryEndpoint endpoint` when starting the TAO Naming Service server to specify the multicast group it is to join and on which port it is to listen for multicast requests. Use it when starting clients of the TAO Naming Service to specify the multicast group and port to use when multicasting a TAO Naming Service discovery request. This option is only available on platforms that support IP multicast. Both server and client applications may use this option.

In development situations where there is a limitation as to the port(s) a process is allowed to use, `-ORBMulticastDiscoveryEndpoint address:port` may be useful by allowing multiple Naming Services to listen on the same port, but have different multicast IP addresses.

This option has the following format:

```bash
-ORBMulticastDiscoveryEndpoint address:port
```
where address is the Naming Service’s multicast group IP address. This must be a class D address in the range 224.0.0.0 to 239.255.255.255. The Naming Service will fail if it is passed an address outside of this range.

port is the Naming Service’s multicast request port number. This port number must be valid for your operating system and user permissions.

See Also 19.8.20, 24.2

Example This example shows how to use the -ORBMulticastDiscoveryEndpoint command line option to specify the multicast request group and port to use:

   Naming_Service -ORBMulticastDiscoveryEndpoint 224.1.1.1:9999
   myserver -ORBMulticastDiscoveryEndpoint 224.1.1.1:9999
   myclient -ORBMulticastDiscoveryEndpoint 224.1.1.1:9999

Alternatively the clients can use -ORBInitRef or -ORBDefaultInitRef to specify the multicast endpoint:

   myserver -ORBInitRef NameService=mcast://224.1.1.1:9999/NameService
   myclient -ORBDefaultInitRef mcast://224.1.1.1:9999
19.8.20 **ORBNameServicePort port**

**Description** When CORBA::ORB::resolve_initial_references("NameService") is called, TAO, by default, uses IP multicast to find the TAO Naming Service. It is often the case that more than one Naming Service is running on a local network during development, or in a deployed application where a large number of objects are logically partitioned in some way. In such a case, it is undesirable for all of the Naming Service servers to be listening for multicast requests on the same port.

At startup, the TAO Naming Service is assigned a port on which to listen for multicast requests. The –ORBNameServicePort option allows the user to specify this port. If this option is not used, then the NameServicePort environment variable (if it is set) is used. Otherwise, if this option is not used and the environment variable is not set, then the port number is set to the value of TAO_DEFAULT_NAME_SERVER_REQUEST_PORT, defined in $TAO_ROOT/tao/default_ports.h as 10013.

**Usage** Use –ORBNameServicePort when starting the TAO Naming Service to specify on which port it is to listen for multicast requests. Use it when starting clients of the Naming Service to tell them the multicast port to use when multicasting a TAO Naming Service discovery request. This option is only available on platforms that support IP multicast. Both server and client applications may use this option.

This option has the following format:

```
-ORBNameServicePort port
```

where `port` is the Name Server’s multicast request port number. This port number must be valid for your operating system and user permissions.

**See Also** 19.8.19, 24.2

**Example** The following example shows how to use the –ORBNameServicePort command line option to specify the multicast request port to use:

```
Naming_Service -ORBNameServicePort 12345
myserver -ORBNameServicePort 12345
myclient -ORBNameServicePort 12345
```

which is equivalent to:
ORB Initialization Options

Naming_Service -ORBMulticastDiscoveryEndpoint 224.9.9.2:12345
myserver -ORBMulticastDiscoveryEndpoint 224.9.9.2:12345
myclient -ORBInitRef NameService=mcast://:12345/NameService
19.8 Option Descriptions

19.8.21 ORBNegotiateCodesets enabled

Values for enabled

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disables the use of the codeset negotiation feature for character and wide character data</td>
</tr>
<tr>
<td>1 (default)</td>
<td>Enables codeset negotiation and possibly character and wide character data translation</td>
</tr>
</tbody>
</table>

Description

Codeset negotiation is a tool used by CORBA applications to determine the numerical codes used to represent character data. Codeset negotiation enables applications running on systems that use different character representations, such as UTF-8 or Latin1, to accurately exchange character data, by having one side or the other translate the character codes so that the same character is represented.

There are may instances where this functionality is simply unused, for instance when the peer applications are all using TAO and running on identical or similar hardware, with similarly configured operating systems. In fact, applications that use TAO often operate on embedded systems that would benefit from not even loading the code to support codeset negotiation and character translation into memory.

The default value of this option for dynamically linked applications is 1. Statically linked applications that do not explicitly link to and initialize the codeset library will behave as though -ORBNegotiateCodesets 0 was set.

Usage

All of the functional code for supporting codeset negotiation resides in a separately loadable library called TAO_Codeset. When the -ORBNegotiateCodesets option is enabled in a dynamically linked application, this library is loaded automatically.

Statically linked applications that want to allow configuration of codeset negotiation must explicitly link to the TAO_Codeset library and must also include the initializer code somewhere in the application with:

```c
#include "tao/Codeset/Codeset.h"
```

You must enable the MPC feature `negotiate_codesets` to have static applications link to the TAO_Codeset library. For example, you may set the following option in

```bash
$ACE_ROOT/bin/MakeProjectCreator/config/default.features:
```
ORB Initialization Options

negotiate_codesets=1

TAO will report an error, "Unable to load TAO_Codeset", if the -ORBNegotiateCodesets option is set and the TAO_Codeset library is not available. If this occurs, the application will continue to run, but without codeset support.

See Also 20.2.8, 4.3.2.3

Example The following example shows how to disable the use of codeset negotiation:

```bash
myserver -ORBNegotiateCodesets 0
```
19.8.22 ORBNodelay enabled

Values for enabled

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disables TCP_NODELAY, thus enabling the Nagle algorithm and introducing a delay into TCP packet sends.</td>
</tr>
<tr>
<td>1</td>
<td>(default) Eliminates the delay in packet sends introduced by the Nagle algorithm.</td>
</tr>
</tbody>
</table>

Description

TAO enables the TCP_NODELAY socket option by default, resulting in the disabling of the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets on a WAN. Using -ORBNodelay with an argument of 0 disables the TCP_NODELAY socket option and thus enables the Nagle algorithm.

Usage

Applications with many small request/reply messages and stringent timing requirements can increase performance and improve predictability by eliminating the delay in TCP packet sends introduced by the Nagle algorithm.

See Also


Example

The following example shows how to disable TCP_NODELAY, thereby enabling the Nagle algorithm:

```
myserver -ORBNodelay 0
```
19.8.23  ORBNoProprietaryActivation

Description  This is a new CORBA 3 ORB initialization option. It indicates that a server should avoid the use of any proprietary activation framework upon start up. Registration with an Implementation Repository (IMR) is an example of such proprietary activation framework behavior that could be performed. Future implementations may provide additional behaviors, based on CORBA 3’s object reference template (ORT) specification.

Note  This option is not currently supported in TAO. However, if this option is present in the argument list (argv[]), CORBA::ORB_init() will accept the option, but raises the CORBA::NO_IMPLEMENT system exception. Also, TAO currently requires that this option, if present, must be followed by an argument (e.g., -ORBNoProprietaryActivation 1), otherwise CORBA::ORB_init() will not correctly parse the remaining options.

Usage  This option should not be used in the current version of TAO. It is described here only because some ORB initialization code exists to parse this option.

See Also  19.8.35, Section 4.5.1.3 of the CORBA 3 specification (OMG Document formal/02-12-06)
19.8.24 ORBObjRefStyle style

Values for style

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IOR (default)</td>
<td>Use the standard OMG IOR format.</td>
</tr>
<tr>
<td>URL</td>
<td>Use a more readable format, similar to the form used in Universal Resource Locators (URLs).</td>
</tr>
</tbody>
</table>

Description

This option specifies the format used for printing Interoperable Object References (IORs) as strings. Stringified IORs can be printed using the standard OMG IOR format (“IOR:” followed by a long string of hexadecimal digits) or the more readable Universal Resource Locators (URL)-like format. OMG IOR format is the default.

Usage

During development and testing, you will often want to print IORs using the URL-like format for easier debugging. However, the OMG standard IOR format is slightly more efficient when used with operations such as CORBA::ORB::string_to_object().

URL-style object references are provided in a format that is compatible with the OMG-specified corbaloc format. See the description of -ORBInitRef in 19.8.13 for more information on corbaloc object references.

In TAO, UIOP (local IPC) URL-style object references have a similar syntax, as follows:

```
<uioploc> = "uioploc://"[<addr_list>]"|"<key_string>
<addr_list> = [<address> "",]* <address>
<address> = [<version> <rendezvous point>]
<rendezvous point> = Valid Filesystem Path
<version> = <major> "." <minor> "@" | empty_string
<major> = number
<minor> = number
<key_string> = <string> | empty_string
```

IORs printed using the URL-like format will have the following form:

```
protocol://V.v@endpoint[/poa_name/...]/ObjectId
```

where V.v represents the major and minor protocol version.

For example:

```
corbaloc:iiop:1.2@host.domain.com:46432/NameService
uioploc://1.2@/tmp/TAOBZMY5Y|NameService
```
Note  

TAO uses a vertical bar ‘|’ in UIOP URL-style object references to separate the object key from the rendezvous point. The forward slash ‘/’ could not be used because it is a valid character in both the object key and the rendezvous point.

For transient object references, a time stamp is encoded within the IOR as well.

Both server and client applications may use this option.

Example  
The following example shows how to use the -ORBObjRefStyle command line option to specify that IORs should be displayed using a URL-like format:

myserver -ORBObjRefStyle URL
19.8.25 ORBPreferredInterfaces *list*

**Description**  
This option allows a client ORB to be configured during the initialization with the capability to choose preferred interface(s) during the invocation phase. This capability is very useful for clients on multi-homed hosts because it allows the client to choose specific interfaces/networks with which to communicate with remote targets.

**Note**  
Not all transport protocols use sockets. Thus, the arguments provided to these options may be interpreted in a protocol-specific way.

**Usage**  
This option is only significant for ORBs in a client role. The *list* parameter may have the following format:

```
target_network:local_network,...
```

Both the `target_network` and the `local_network` parameters may be specified as host names, IP addresses, or the wild-card character `*`.

**See Also**  
19.8.11

**Examples**  
Suppose a client is running on a host with two network interface cards with the following host names:

```
malory.ociweb.com
arthur.ociweb.com
```

Now, suppose this client needs to communicate with a (target) server running on a host named `www.sometargethost.com`. The application wishes to constrain all connections to the server on that target host through the interface `malory.ociweb.com`.

The following example shows how to use the `-ORBPreferredInterfaces` command line option to achieve the desired result:

```
myclient -ORBPreferredInterfaces "www.sometargethost.com:malory.ociweb.com"
```

If all connections with remote servers at `sometargethost.com` are to be constrained to the `malory.ociweb.com` interface, the wild-card character `*` could be used, as in the following example:
ORB Initialization Options

```
myclient -ORBPreferredInterfaces "*.sometargethost.com:malory.ociweb.com"
```
19.8.26 ORBRcvSock **buffersize**

**Description**
You may find it useful, when tuning an application’s performance, to control the sizes of the protocol’s receive buffer in an attempt to maximize throughput. This can be especially helpful if you have *a priori* knowledge of the application’s needs and the dynamics of the network in which the application is operating. The `-ORBRcvSock` option allows you to control the buffer size of the socket on the receiving side.

**Note**
Not all transport protocols use sockets. Thus, the arguments provided to these options can be thought of as general I/O buffer sizes that may be interpreted in a protocol-specific way.

**Usage**
Both server and client applications may use this option. The default value for this option, `ACE_DEFAULT_MAX_SOCKET_BUFSIZE`, is defined as 65536 in `$ACE_ROOT/ace/OS.h`. Specify a value in bytes that is less than 65536. Larger values generally improve throughput.

**See Also**
19.8.30

**Example**
The following example shows how to use the `-ORBRcvSock` command line option to specify receiving socket buffer size:

```
myserver -ORBRcvSock 8192
```
**ORB Initialization Options**

### 19.8.27 ORBServerId server_id

**Description**
The CORBA 3 specification (OMG Document formal/02-12-06) defines an ORB initialization option to uniquely identify a server to an Implementation Repository (IMR). The `-ORBServerId` option accepts a string for the `server_id` argument. All object reference templates created in this ORB will return the specified server id in the `server_id` attribute. All ORBs created in a server must share the same server id. The default server id in TAO is an empty string.

**Usage**
This option currently has little effect on TAO in terms of how servers interact with the IMR. The ORB’s server id is, however, used in the creation of an object reference template by each new POA that is created in a server. Future revisions may assign more behavior to the server id specified by this option.

**See Also**
19.8.35, Section 4.5.1.1 of the CORBA 3 specification (OMG Document formal/02-12-06)

**Example**
The following example shows how to use the `-ORBServerId` command line option to control the server id of the ORB.

```
myserver -ORBServerId MyServer
```
19.8.28 ORBServiceConfigLoggerKey *logger_key*

**Description**  This option allows you to specify where to write ORB logging output. This option is equivalent to passing the key to the `ACE_Service_Config::open()` function.

**Usage**  Both server and client applications may use this option. The default value is `ACE_DEFAULT_LOGGER_KEY`, which depends on the platform type. Platforms that support stream pipes use `/tmp/server_daemon`. Others use `localhost:20012`.

**Example**  The following will cause logging output to be written to the specified port on the local machine.

```bash
myserver -ORBServiceConfigLoggerKey localhost:9999
```
### ORBSingleReadOptimization enabled

#### Values for enabled

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disables the single read optimization, thereby requiring two read operations to read each GIOP message (one to read the GIOP message header, one to read the message body).</td>
</tr>
<tr>
<td>1 (default)</td>
<td>Each time the ORB reads a GIOP message from the transport, it will read <code>TAO_MAXBUFSIZE</code> bytes, hoping to read an entire message. If it reads more than one message (including a partial message), it will queue them for later processing.</td>
</tr>
</tbody>
</table>

#### Description

A GIOP message includes a fixed-size message header and a message body. A field in the message header indicates the size of the message in bytes. An ORB can always read exactly one GIOP message by using the following algorithm:

- Read the fixed-size message header.
- Get the message size from the message header.
- Read the rest of the message as indicated by the size field in the header.

However, performing two read operations for each message may not be the most efficient use of resources. By attempting to always read a larger chunk of bytes and queuing any extra messages (or partial message) for later processing, performance can be greatly improved.

The single read optimization in TAO enables the ORB to attempt to read each message in a single read operation rather than in two read operations.

#### Usage

Both server and client applications may use this option.

Normally, the default value of 1 (true) for this option is appropriate because it results in better performance since fewer read operations are performed during request and reply processing.

In real-time applications, this option should be set to 0 (false) to disable the single read optimization, otherwise it may lead to priority inversions. For example, suppose two or more requests arrive at a socket and are held in the socket’s receive buffer for reading. If multiple requests are read from the socket in a single read, the first request will be processed and the rest of the requests will be queued. A reactor notification will then be used to wake up a follower thread in the ORB’s leader-followers model. Meanwhile, new higher priority requests may arrive on other sockets. But, since the TP reactor (the default reactor type in TAO) dispatches notifications before normal I/O, the
lower-priority queued messages will be processed \textit{before} newly-arrived higher-priority requests, thereby leading to a priority inversion.

\textbf{See Also}  Chapter 9

\textbf{Example}  The following example shows how to disable TAO’s single read optimization:

\begin{verbatim}
myserver -ORBSingleReadOptimization 0
\end{verbatim}
ORB Initialization Options

19.8.30 ORBSndSock buffersize

Description You may find it useful, when tuning an application’s performance, to control the sizes of the protocol’s send buffer in an attempt to maximize throughput. This can be especially helpful if you have a priori knowledge of the application’s needs and the dynamics of the network in which the application is operating. The -ORBSndSock option allows you to control the buffer size of the socket on the sending side.

Note Not all transport protocols will use sockets. Thus, the arguments provided to these options can be thought of as general I/O buffer sizes that may be interpreted in a protocol-specific way.

Usage Both server and client applications may use this option. The default value for this option, ACE_DEFAULT_MAX_SOCKET_BUFSIZE, is defined as 65536 in $ACE_ROOT/ace/OS.h. Specify a value in bytes that is less than 65536. Larger values generally improve throughput.

See Also 19.8.26

Example The following example shows how to use the -ORBSndSock command line option to specify the sending socket buffer size.

myserver -ORBSndSock 4096
19.8 Option Descriptions

19.8.31 ORBStdProfileComponents enabled

Values for enabled

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (default)</td>
<td>The ORB generates the OMG standard profile components.</td>
</tr>
<tr>
<td>0</td>
<td>The ORB does not generate the OMG standard profile components.</td>
</tr>
</tbody>
</table>

Description

Agents that accept object requests or that provide locations for objects (i.e., servers) publish object locations using opaque, protocol-specific profiles. The standard IORs encapsulate these profiles. A single IOR may contain multiple profiles for a single CORBA object. For example, an IIOP profile includes the host name and port at which the server ORB listens for requests.

A profile may also contain, in its so-called tagged components, additional information, such as the character set understood by the server, security tokens, and priority information. These tagged components are optional, but if present, must be retained as part of the IOR, even if the IOR is passed between ORBs. The default is to generate the optional standard profile components.

This option controls generation of the following optional tagged components:

- TAG_ORB_TYPE
- TAG_CODE_SETS

Usage

Use the -ORBStdProfileComponents option to control whether the ORB will generate optional tagged components in IORs. This option is applicable only to server applications.

Specify a value of 0 for this option to suppress generation of the optional tagged components, if clients do not need them, are operating under severe memory constraints, or anticipate a very large number of IORs.

Impact

Suppressing the generation of these optional tagged components saves 28 bytes per IOR. Since these components are optional, the use of this option to suppress them does not impact interoperability with other GIOP 1.1- or 1.2-compliant ORBs.

See Also

The definition of TAO_STD_PROFILE_COMPONENTS in $TAO_ROOT/tao/orbconf.h.

Example

The following example shows how to use the -ORBStdProfileComponents command line option to instruct the ORB not to generate optional standard profile components:

```bash
myserver -ORBStdProfileComponents 0
```
ORB Initialization Options

19.8.32 ORBSvcConf config_file_name

Description
By default, a service-configurator-based application looks in the current
directory for a file named svc.conf, which supplies directives to the service
configurator. You can use the -ORBSvcConf option to specify a different file
name. The config_file_name argument can be any valid path. The target file
may contain directives for non-TAO-related service objects and for TAO
components. You can supply multiple files to the service configurator by
supplying multiple -ORBSvcConf options on the command line. The
directives in each file are supplied to the service configurator in an additive
fashion. Using -ORBSvcConf on the command line is equivalent to passing
the -f option to the open() function of the ACE_Service_Config class.

Usage
Both server and client applications may use this option. It is only applicable
on platforms with file systems.

A common usage for this option is when a directory contains several service-
configurator-based applications. Each needs its own service configuration file,
so it is not possible to use the default file name svc.conf. Use this option to
supply a different configuration file to each application. You may also use it to
supply different configurations to separate instances of the same application.

See Also
18.4

Example
The following examples show how to use the -ORBSvcConf command line
option to specify alternate service configuration files to applications:

```bash
myserver -ORBSvcConf tsscommon.conf
myserver -ORBSvcConf $TESTROOT/rt_test/svc.conf -ORBSvcConf $HOME/svc.conf
```
19.8.33 ORBSvcConfDirective *directive*

**Description** Use `-ORBSvcConfDirective` to supply a single directive to the service configurator from the command line. This option may be repeated to pass multiple directives. The *directive* argument can be any valid service configurator directive string. Using `-ORBSvcConfDirective` on the command line is equivalent to passing the `-S` option to the `open()` function of the `ACE_Service_Config` class.

**Usage** Both server and client applications may use this option. When it is not possible to use a service configuration file with an application, applications may use this option. A good example is an environment where there is no file system. Developers may also use this option during development and testing to quickly try alternate configurations, without modifying the configuration file.

**See Also** 18.4, 19.8.32

**Example** The following example shows how to use the `-ORBSvcConfDirective` command line option to pass directives to the ORB’s service configurator:

```
myserver -ORBSvcConfDirective "static Resource_Factory '-ORBReactorType tp'"
```
19.8.34 ORBTradingServicePort port

Description When CORBA::ORB::resolve_initial_references("TradingService") is called, TAO, by default, uses IP multicast to find the TAO Trading Service. It is often the case that more than one Trading Service is running on a local network during development, or in a deployed application where a large number of objects are logically partitioned in some way. In such a case, it is undesirable for all of the Trading Service servers to be listening for multicast requests on the same port.

At startup, the TAO Trading Service is assigned a port on which to listen for multicast requests. The -ORBTradingServicePort option allows the user to specify this port. If this option is not used, then the TradingServicePort environment variable (if it is set) is used. Otherwise, if this option is not used and the environment variable is not set, then the port number is set to the value of TAO_DEFAULT_TRADING_SERVER_REQUEST_PORT, defined in $TAO_ROOT/tao/default_ports.h as 10016.

Usage Use -ORBTradingServicePort when starting the TAO Trading Service to specify on which port it is to listen for multicast requests. Use it when starting clients of the Trading Service to tell them the multicast port to use when multicasting a TAO Trading Service discovery request. This option has the following format:

-ORBTradingServicePort port

where port is the Trading Service’s multicast request port number. This port number must be valid for your operating system and user permissions.

Both server and client applications may use this option. This option is only available on platforms that support IP multicast.

Example The following example shows how to use the -ORBTradingServicePort command line option to specify the multicast request port to use:

$TAO_ROOT/orbsvcs/Trading_Service/Trading_Service -ORBTradingServicePort 12345
myserver -ORBTradingServicePort 12345
19.8.35 ORBUseIMR enabled

Values for enabled

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (default)</td>
<td>Do not use the Implementation Repository (ImplRepo).</td>
</tr>
<tr>
<td>1</td>
<td>Use the ImplRepo.</td>
</tr>
</tbody>
</table>

Description

The use of the -ORBUseIMR option has the following effects:

- In the ImplRepo, each of the server’s persistent POAs has its address set to the server’s address and its current execution status set to running.
- The server embeds the ImplRepo’s address, rather than its own address, in the IORs of its persistent objects.

Usage

In order for a server to use the ImplRepo,

- The name of each of its persistent POAs must be added to the ImplRepo.
- It must obtain an object reference to the ImplRepo.
- It must be started using the -ORBUseIMR option with a value of 1.

The environment variable TAO_USE_IMR can be used instead of passing -ORBUseIMR option on a server’s command line. Also, if the ImR_ACTIVATOR is used to start a server, the -ORBUseIMR option is automatically added to the server’s command line options by the Activator.

See Also

Chapter 30

Example

If the command

ImplRepo_Service -o implrepo.ior

is used to start the ImplRepo, the command:

tao_imr -ORBInitRef ImplRepoService=file://implrepo.ior add MyPoa

is used (see 30.2) to add the name MyPoa to the repository, and if MyServer is a server with a single persistent POA named MyPoa, the command

MyServer -ORBUseIMR 1 -ORBInitRef ImplRepoService=file://implrepo.ior

causes the following to occur:

- MyServer is started.
ORB Initialization Options

- In the ImplRepo, MyPoa’s address is set to MyServer’s address.
- In the ImplRepo, MyPoa’s current execution status is set to running.
- The address of the ImplRepo that was obtained from implrepo.ior is used to construct IOR’s for MyPoa’s objects.
19.8.36 **ORBUseSharedProfile enabled**

Values for *enabled*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Specifies that the ORB should create a new profile for each endpoint when building profiles for IORs.</td>
</tr>
<tr>
<td>1 (default)</td>
<td>Specifies that the ORB will combine multiple endpoints (with the same protocol and protocol version) into a single profile.</td>
</tr>
</tbody>
</table>

**Description**  
When an IOR is created, the endpoints on which the ORB is listening for requests are encoded into the IOR. The endpoints are contained within one or more *profiles* in the IOR. An IOR may have multiple profiles and each profile may have multiple endpoints. This option controls whether or not TAO will combine multiple endpoints having the same protocol and protocol version (e.g., IIOP 1.2) into a single profile.

**Usage**  
Use the `-ORBUseSharedProfile` option to control whether the ORB will combine multiple endpoints into a single profile in IORs. This option is applicable only to server applications.

**Impact**  
Combining multiple endpoints into a single profile results in slightly smaller IORs. There is no appreciable performance impact associated with the use of this option. Some legacy client ORBs may not be able to access more than one endpoint from a single profile.

**See Also**  
19.8.10, 19.8.17

**Example**  
myserver -ORBUseSharedProfile 0
19.8.37 ORBVerboseLogging level

Values for level

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (default)</td>
<td>The same as not using the option.</td>
</tr>
<tr>
<td>1</td>
<td>Adds a timestamp to prefix each line of code.</td>
</tr>
<tr>
<td>2</td>
<td>Add even more verbosity</td>
</tr>
</tbody>
</table>

Description
This option controls the amount of status data printed on each line of the debug log. Higher numbers generate more output.

Usage
Use the -ORBVerboseLogging option to obtain additional timing information about each debug log entry.

Impact
The additional information printed out for each log entry may decrease your application performance.

See Also
19.8.6, 19.8.18

Example
myserver -ORBVerboseLogging 1 -ORBDebugLevel 10
CHAPTER 20

Resource Factory

20.1 Introduction

The resource factory is responsible for constructing and providing access to various resources used by ORBs within both clients and servers. ORB resources include a reactor, protocol factories, and allocators for buffers used during encoding and decoding of data transmitted between ORBs. Protocol factories are responsible for supplying acceptors, connectors, and connection handlers to enable interprocess communication.

TAO provides several different types of resource factories, summarized in Table 20-1. The choice of resource factory to use depends upon such considerations as the features your application requires and the environment in which it will run. Many applications will function just fine with the default resource factory and will not require any of the special features found in the alternate resource factory implementations.
With the exception of the advanced resource factory, the resource factory used by the ORB is registered with the service configurator as Resource_Factory. TAO’s default resource factory is statically registered with the service configurator, so the static directive is used to supply initialization options to it. To change the behavior of the default resource factory, add a line similar to the line below to your service configuration file:

```
static Resource_Factory "-ORBoption value -ORBoption value ..."
```

The service configurator is discussed in greater detail in 18.3. The options supported by the default resource factory are discussed in the following pages.

## 20.2 Interface Definition

Within a TAO application, the resource factory is accessed via an interface defined by the base class `TAO_Resource_Factory`. Here we show all of the operations of the resource factory and describe their use in relationship to the ORB, as well as the behavior of the default resource factory implementation provided with TAO.

Whereas a reference to the resource factory may be obtained through the service configurator, it is not intended for use external to the ORB core.

### Table 20-1 Resource Factories provided with TAO

<table>
<thead>
<tr>
<th>Resource Factory</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Resource Factory</td>
<td>20.2</td>
<td>Unless configured otherwise, this is the resource factory used.</td>
</tr>
<tr>
<td>Qt Resource Factory</td>
<td>20.3</td>
<td>This is a specialized resource factory providing the means for integrating with the Qt GUI toolkit from Trolltech.</td>
</tr>
<tr>
<td>Xt Resource Factory</td>
<td>20.4</td>
<td>This is a specialized resource factory providing the means for integrating with the X Window System’s Xt Intrinsics toolkit.</td>
</tr>
<tr>
<td>Advanced Resource Factory</td>
<td>20.5</td>
<td>This factory provides more advanced configuration options in addition to all the features of the default resource factory.</td>
</tr>
</tbody>
</table>
TAO’s ORB core object supplies the interface for retrieving objects supplied by the resource factory.

**Synopsis**
```cpp
class TAO_Export TAO_Resource_Factory : public ACE_Service_Object {
    enum Purging_Strategy {
        LRU,
        LFU,
        FIFO,
        NOOP
    };

    enum Resource_Usage {
        TAO_EAGER,
        TAO_LAZY
    };

    TAO_Resource_Factory (void);
    virtual ~TAO_Resource_Factory (void);
    virtual void disable_factory (void) = 0;
    virtual ACE_Reactor* get_reactor (void);
    virtual void reclaim_reactor (ACE_Reactor* reactor);
    virtual TAO_LF_Strategy* create_lf_strategy (void) = 0;
    virtual TAO_Acceptor_REGISTRY* get_acceptor_registry (void);
    virtual TAO_Connector_REGISTRY* get_connector_registry (void);
    virtual ACE_Allocator* input_cdr_dblock_allocator (void);
    virtual ACE_Allocator* input_cdr_buffer_allocator (void);
    virtual ACE_Allocator* input_cdr_msgblock_allocator (void);
    virtual ACE_Allocator* output_cdr_dblock_allocator (void);
    virtual ACE_Allocator* output_cdr_buffer_allocator (void);
    virtual ACE_Allocator* output_cdr_msgblock_allocator (void);
    virtual int input_cdr_allocator_type_locked (void);
    virtual ACE_Allocator* output_cdr_dblock_allocator (void);
    virtual ACE_Allocator* output_cdr_buffer_allocator (void);
    virtual ACE_Allocator* output_cdr_msgblock_allocator (void);
    virtual int use_locked_data_blocks (void) const;
    virtual TAO_Flushing_Strategy* create_flushing_strategy (void) = 0;
    virtual TAO_Connection_Purging_Strategy* create_purging_strategy (void) = 0;
    virtual int purge_percentage (void) const;
    virtual int cache_maximum (void) const;
    virtual int max_muxed_connections (void) const;
    virtual int locked_transport_cache (void);
    virtual ACE_Allocator* create_corba_object_lock (void);
    virtual TAO_Resource_Factory::Resource_Usage resource_usage_strategy (void) const = 0;
};
```
20.2.1 Reactor Management

The resource factory interface defines an operation that returns a pointer to a newly-created reactor. Each ORB instance in an application will have a reactor, and may even have multiple reactors, but no reactor will be shared among ORBs.

```cpp
virtual ACE_Reactor* get_reactor (void);
```

TAO will use a reactor when dealing with connection management involving the predefined protocols, for example IIOP, UIOP, and SHMIOP. At the very least, TAO uses the reactor to supply non-blocking event-handling behavior in the ORB. The interface also defines an operation that reclaims or deallocates any resources used by the reactor.

```cpp
virtual void reclaim_reactor (ACE_Reactor* reactor);
```

In addition to the default reactor, TAO provides various other reactors for specialized needs. Specialized reactors are used with the Qt and Xt resource factories. Also, the advanced resource factory allows the selection of other TAO-provided reactors if the default does not suit your needs. See 20.5 for information on using the advanced resource factory and selecting other reactors.

Another operation that relates to the reactor is `create_lf_strategy()`:

```cpp
virtual TAO_LF_Strategy* create_lf_strategy (void) = 0;
```

The `TAO_LF_Strategy` returned by this operation tells the ORB the method by which threads vie for access to the reactor. The default resource factory always returns a valid strategy, so the leader-follower model will be used to synchronize access to the reactor. The advanced resource factory will return a valid leader-follower strategy unless the reactor type chosen is `select_st`, in which case it will return a null strategy.

An advanced resource factory option, shown in table Table 20-2, is used to specify the type of reactor to be used by the ORB.
### 20.2.2 Allocators for CDR Conversion

Before data is transferred between ORBs, either as parameters to an operation or as the result from an operation, the data must be put into a form that is suitable for transmission. TAO is responsible for encoding the data on the sending side and decoding the data on the receiving side. Encoding the data involves **marshaling**, which is gathering up the bits of information to be transmitted and placing them in a predictable order. The predictable order specified by GIOP is called the Common Data Representation, or CDR. The CDR is discussed in detail in *Advanced CORBA Programming with C++*, 13.3.

When TAO encodes data for sending or decodes data received, space is required to hold the intermediate form of the data. This interim space is managed using **ACE_Message_Blocks**. Message blocks contain data in units of **ACE_Data_Blocks**, which provide a reference-counted wrapper around the actual data buffer. The space used by these buffers is owned by the ORB that manipulates them, therefore the allocation strategy used to manage the space is provided by the resource factory. Different allocators can be used for **ACE_Message_Blocks**. In addition, the **ACE_Message_Block** provides the ability to use separate allocators for the **ACE_Data_Blocks** and the buffers. Therefore, the resource factory interface provides operations used by the ORB to access allocators used to create **ACE_Message_Block** and **ACE_Data_Block** objects and data buffers for input and output. The input CDR allocator accessor are:

```cpp
virtual ACE_Allocator* input_cdr_msgblock_allocator (void);
virtual ACE_Allocator* input_cdr_dblock_allocator (void);
virtual ACE_Allocator* input_cdr_buffer_allocator (void);
```

The buffers used to receive and decode incoming data may be passed up the processing chain from the actual communication endpoint, through the ORB, to the application code responsible for dealing with the data. Locking may be
required if multiple threads will access the data. The locked input CDR allocator is:

    virtual int input_cdr_allocator_type_locked (void);

The advanced resource factory option `-ORBInputCDRAllocator` (see 20.7.4) is provided to specify the type of lock used to protect access to the input CDR allocators and the data they allocate. See 20.5 for information on using the advanced resource factory. The default resource factory always returns a non-zero value, which means access to the allocator for creating and destroying data buffers is synchronized and therefore safe to be used by many threads.

The following operations are used by the ORB to access allocators used for encoding outgoing data:

    virtual ACE_Allocator* output_cdr_msgblock_allocator (void);
    virtual ACE_Allocator* output_cdr_dblock_allocator (void);
    virtual ACE_Allocator* output_cdr_buffer_allocator (void);

Access to the `ACE_Data_Block`s used to encode data being transmitted does not normally need to be synchronized since the data blocks are not used for any application-level processing. The ORB core provides data blocks used during the CDR conversion process and uses the following resource factory operation to determine the type of data block it is expected to produce:

    virtual int use_locked_data_blocks (void) const;

The default resource factory’s implementation returns a non-zero value from `use_locked_data_blocks()` when `-ORBConnectionCacheLock` is set to `thread` and returns zero when `-ORBConnectionCacheLock` is set to `null`. The `-ORBConnectionCacheLock` option also affects connection caching. (See 20.2.4 for a discussion of connection caching in TAO.)

In addition, the advanced resource factory’s implementation returns a non-zero value from `use_locked_data_blocks()` when `-ORBInputCDRAllocator` is set to `thread`, and returns zero when `-ORBInputCDRAllocator` is set to `null`.

Since `-ORBConnectionCacheLock` and `-ORBInputCDRAllocator` are separate options, both of which affect data-block locking, it is possible to provide conflicting values when using the advanced resource factory. If
conflicting values are given, the value provided by -ORBConnectionCacheLock will be used for data-block locking. If you use both of these options, they should have the same setting to prevent unexpected results.

An advanced resource factory option, shown in table Table 20-3, is used to control the type of locking used for accessing the input CDR conversion allocator.

Table 20-3 Options for Configuring Input CDR Conversion Allocator

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBInputCDRAllocator {thread</td>
<td>20.7.4</td>
<td>Specify the type of locking to be used for accessing the input CDR conversion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>allocator.</td>
</tr>
</tbody>
</table>

20.2.3 Message Flushing Strategies

Data sent from one ORB to another must be converted into a common data representation (CDR). After CDR conversion, the data is stored in ACE_Message_Block. The message blocks are queued until they can be flushed to the outgoing transport layer. The strategy by which messages are flushed can vary, and affects the timing of the data being sent. The resource factory interface defines the following operation, which returns the strategy to be used for flushing messages:

```
virtual TAO_Flushing_Strategy* create_flushing_strategy (void) = 0;
```

The default flushing strategy returned by the default resource factory uses the leader-follower pattern. Table 20-4 shows the option that affects which message flushing strategy is used by the ORB.

Table 20-4 Message Flushing Strategy Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBFlushingStrategy {leader</td>
<td>20.6.6</td>
<td>Selects the strategy for how messages</td>
</tr>
<tr>
<td>follower</td>
<td>reactive</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20.2.4 Connection Cache Management Strategies

A client ORB establishes at least one connection to every server with which it communicates. Under certain circumstances, more than one connection might be established between a given client and a given server. For example, you can
configure the client-side ORB such that it will not multiplex requests over a single connection (see 22.3.3), or you can establish multiple connections to a server where each connection is used to transmit requests of a certain priority or range of priorities (see 9.3.10).

The ORB maintains connections in a cache. When a request is about to be sent, the ORB looks for an existing connection in the cache to use to transmit the message. If a suitable connection is not available, the ORB may establish a new connection for the message, then add the new connection to the cache. TAO relies on these cached connections in both the server and the client for optimization. As connections accumulate in the cache, the cache may need to be purged to allow new connections to be added.

The TAO_Acceptor and TAO_Connector both use cache management (purging) strategies supplied by the resource factory. The following operations of the resource factory interface deal with cache management or purging:

```
enum Purging_Strategy {
    LRU,
    LFU,
    FIFO,
    NOOP
};

virtual int locked_transport_cache (void);
virtual TAO_Connection_Purging_Strategy* create_purging_strategy (void) = 0;
virtual int purge_percentage (void) const;
virtual int cache_maximum (void) const;
virtual int max_muxed_connections (void) const;
virtual ACE_Lock* create_cached_connection_lock (void);
```

There are four defined purging strategies used by these operations:

- **least recently used** (LRU)—this strategy removes connections that have been idle the longest.
- **least frequently used** (LFU)—this strategy removes connections that are used least often.
- **first in first out** (FIFO)—this strategy removes the oldest connections, regardless of how often they are used.
- **no operation** (NOOP)—this strategy does not purge connections at all.
The `create_purging_strategy()` operation creates and returns a `TAO_Connection_Purging_Strategy` object that will match one of the above-defined strategies. The `TAO_Connection_Purging_Strategy` object is passed to the `TAO_Transport_Cache_Manager`, thereby instructing it on how to purge the cache. The default resource factory uses the LRU strategy exclusively. The advanced resource factory can return any one of these strategies as instructed by the `-ORBConnectionPurgingStrategy` option. See 20.5 for information on using the advanced resource factory.

The `purge_percentage()` operation indicates the quantity of the cache to remove if conditions require purging. The value returned must be between 0 and 100. The default resource factory returns 20 as the purge percentage by default.

The `cache_maximum()` operation returns the maximum number of connections that may be cached in the ORB. The default value is system dependent and is based on the maximum number of file descriptors available to the process.

The `max_muxed_connections()` operation returns the maximum number of multiplexed connections that are allowed for a single remote endpoint. The default value is zero, meaning there is no theoretical limit on the number of connections that can be established (and cached) for a given remote endpoint. If a value greater than zero is returned, and that limit is reached (i.e., the maximum number of multiplexed connections have already been established to a given endpoint), the thread needing a connection to transmit a request will wait on a condition variable to be awaken when an existing connection becomes idle.

The `create_cached_connection_lock()` operation returns the type of lock that should be used when threads indirectly access the connection cache through the ORB. Concrete, protocol-specific, derived classes of the `TAO_Transport` abstract base class use this lock to ensure thread-safe access to the connection cache when sending a request. The possible lock types returned are a null lock (for no locking, as in single-threaded applications) and a thread-safe lock (for use in multithreaded applications). The default resource factory returns the thread-safe lock by default.

The `locked_transport_cache()` operation returns a boolean value to indicate whether the connection cache needs to have a lock or not. The `TAO_Transport_Cache_Manager` uses the value returned by this operation to determine whether a lock should be used when threads access the
connection cache directly through its interface. The default resource factory’s implementation of this operation returns false when the
-ORBConnectionCacheLock option is set to null and true when this option is set to thread. By default, the default resource factory returns true. Therefore, unless configured otherwise, access to the connection cache is synchronized.

Table 20-5 shows the options that control the cache management behavior of the default resource factory.

Table 20-5 Connection Cache Management Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBConnectionCachePurgePercentage percent</td>
<td>20.6.4</td>
<td>Supply the amount of the cache to purge. The default is 20 percent.</td>
</tr>
<tr>
<td>-ORBConnectionCacheMax limit</td>
<td>20.6.3</td>
<td>Supply the maximum number of connections that may be cached in the ORB. The default is system dependent.</td>
</tr>
<tr>
<td>-ORBConnectionCacheLock</td>
<td>20.6.2</td>
<td>Select the type of lock the ORB will use to access the connection cache. The default is thread.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20.6.8</td>
<td>Supply the maximum number of multiplexed connections allowed per remote endpoint.</td>
</tr>
</tbody>
</table>

20.2.5 Object Key Table Lock

Many object references may share an object key. Rather than repeating this information in every IOR, a table is used to store object keys. The object key table keeps track of the object keys that are generated and made available through IORs. The table manages the lifespan of the object keys within the ORB through reference counting.

In multithreaded applications, the object key table must be locked to protect it from concurrent access. In single-threaded applications, no locking is required.

The resource factory interface provides the following operation that is used by the ORB to access the type of lock used to protect the object key table:

```
virtual ACE_Lock *create_object_key_table_lock (void);
```
20.2 Interface Definition

Table 20-6 shows the option to configure the locking strategy used for the object key table.

**Table 20-6 Object Key Table Locking Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBObjectKeyTableLock</td>
<td>20.6.11</td>
<td>Specify the locking strategy used to control access to the object key table.</td>
</tr>
</tbody>
</table>

### 20.2.6 Protocol Factories

TAO is intended to support more than just the common TCP/IP-based IIOP for communication between ORBs. There are many situations where alternative protocols provide a performance advantage or where TCP/IP is not available. You can enable one or more of the other transport protocols supplied with TAO or you can use a custom protocol.

Since multiple protocols can be enabled and made available to the ORB, the resource factory interface defines operations that initialize and return the protocols the ORB should support. These operations are:

```c
virtual int init_protocol_factories (void);
virtual TAO_ProtocolFactorySet* get_protocol_factories (void);
```

Enabling the use of multiple protocols is a two-step process:

1. Define the protocol by supplying a protocol factory (see 16.4.6).
2. Make the protocol factory available to the ORB.

Table 20-7 shows the option to make a protocol factory available to the ORB. This option can be repeated to make multiple protocol factories available.

**Table 20-7 Protocol Factory Registration Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBProtocolFactory pfactory</td>
<td>20.6.12</td>
<td>Declare that a protocol factory named pfactory is available for supplying connections.</td>
</tr>
</tbody>
</table>

The protocols supplied with TAO are described in Chapter 15.

### 20.2.7 Custom IOR Parsers

CORBA defines a generic format, called Interoperable Object Reference
(IOR), to identify CORBA objects and define one or more paths through which an object can be accessed. Each path references a server location that implements the object and an opaque object key that is valid relative to that particular server’s location.

TAO supports several IOR formats, including corbaloc:, corbaname:, IOR:, and file:. However, some applications may benefit from other formats. For example, an http: IOR format might allow applications to download an object reference from a web server.

However, for a particular IOR format to be used, the ORB must be able to parse it to make it usable to the application. TAO’s resource factory allows application developers to implement custom IOR parsers and dynamically register them with the ORB.

The resource factory interface defines the following operation, which the ORB uses to retrieve the names of the IOR parsers it should use:

```c
virtual int get_parser_names (char**& names, int& number_of_names);
```

The ORB’s string_to_object() operation queries each available IOR parser to see if it can parse the given stringified IOR format. When the ORB finds a match, it uses that parser to convert the string into an object reference. Using an IOR parser is more convenient than adding configuration code in the application’s main() function. It also allows for easier integration with other TAO components, such as the -ORBInitRef option (see 19.8.13).

To implement an IOR parser, you must implement a class derived from TAO_IOR_Parser. The example below shows what an HTTP IOR parser might look like:

```c
class HTTP_Parser : public TAO_IOR_Parser
{
public:
  virtual int match_prefix (const char* ior_string) const;
  virtual CORBA::Object_ptr parse_string (const char* ior,
    CORBA::ORB_ptr orb
    ACE_ENV_ARG_DECL)
    ACE_THROW_SPEC ((CORBA::SystemException));
};
```

The match_prefix() function must recognize all the IOR prefixes that this parser supports. Normally, each IOR parser understands only one prefix. A
20.2 Interface Definition

typical implementation of the `match_prefix()` function might look like this:

```cpp
int HTTP_Parser::match_prefix (const char* ior_string) const
{
    static const char http_prefix[] = "http:";
    int cmp = ACE_OS::strncmp (ior_string, http_prefix, sizeof(http_prefix));
    return (cmp == 0);
}
```

The `parse_string()` function implements the actual string parsing. In your implementation of `parse_string()`, you can safely assume that the string has been validated by the `match_prefix()` function. Typically, `parse_string()` will obtain or construct a string matching one of the predefined IOR formats, such as a corbaloc: Object URL. In this example, the more “normal” IOR is obtained by downloading a document from a web server. It then uses `string_to_object()` on the downloaded IOR string to return the object reference:

```cpp
CORBA::Object_ptr HTTP_Parser::parse_string (
    const char* ior,
    CORBA::ORB_ptr orb
ACE_ENV_ARG_DECL)
ACE_THROW_SPEC ((CORBA::SystemException))
{
    // Parse IOR as if it was an http: URL
    ACE_URL_Addr* url_addr = ACE_URL_Addr::create_addr (ior);

    ACE_HTTP_Addr* http_addr = ACE_dynamic_cast(ACE_HTTP_Addr*,url_addr);

    // Connect to the remote host and download the web page, store the
    // contents in:
    char* document_contents = ...;

    return orb->string_to_object (document_contents ACE_ENV_ARG_PARAMETER);
}
```

TAO uses the ACE Service Configurator framework to find the IOR parsers. See 18.5 for more information on the ACE Service Configurator. The next part of the example shows how to integrate the IOR parser with the service configurator.

First you must declare, in the header file, a factory function and a description of the service, this is easily accomplished via the following ACE macros:
If you are only going to use Unix-like compilers and linkers, then you can simply use TAO in place of Export_Prefix. However, under Microsoft Windows variants, this string must be the prefix of the DLL export/import macros used by your library. If you are going to statically link your IOR Parser into the application you will also need to add the ACE_STATIC_SVC_REQUIRE macro, as follows:

ACE_STATIC_SVC_DECLARE_EXPORT (Export_Prefix, HTTP_Parser)
ACE_FACTORY_DECLARE (Export_Prefix, HTTP_Parser)
ACE_STATIC_SVC_REQUIRE (HTTP_Parser)

Next, you must implement the services defined above. Using another group of helper macros, you should add the following in your source file:

ACE_STATIC_SVC_DEFINE (HTTP_Parser,
    ACE_TEXT("HTTP_Parser"),
    ACE_SVC_OBJ_T,
    &ACE_SVC_NAME (HTTP_Parser),
    ACE_Service_Type::DELETE_THIS |
    ACE_Service_Type::DELETE_OBJ,
    0)
ACE_FACTORY_DEFINE (Export_Prefix, HTTP_Parser)

The second argument to ACE_STATIC_SVC_DEFINE is the name of the service in the ACE Service Configurator. It is customary, but not required, to use the name of the class.

The IOR parsers in the ORB can serve as more complete examples. You might want to look in $TAO_ROOT/tao at FILE_Parser.h, CORBALOC_Parser.h, CORBANAME_Parser.h, DLL_Parser.h, or MCAST_Parser.h.

Finally, you can dynamically add your IOR parser using the -ORBIORParser resource factory option. For example, to add our HTTP IOR parser, we would add the following service configurator directive:

static Resource_Factory "-ORBIORParser HTTP_Parser"

Table 20-8 shows the option that allows adding custom parsers to the resource factory.
20.2 Interface Definition

20.2.8 Code Set Identifiers and Translators

The CORBA specification defines a character set as a finite set of different characters used for the representation, organization, or control of data. Examples of character sets include the English alphabet, Kanji or sets of ideographic characters, corporate character sets (commonly used in Japan), and the characters needed to write certain European languages. A coded character set (or code set) is defined as a set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation or numeric value. Examples include ASCII, ISO 8859-1, JIS X0208 (which includes Roman characters, Japanese hiragana, Greek characters, Japanese kanji, etc.), and Unicode.

If a TAO client or server must operate in a non-US English host environment, it may be necessary to declare the particular code set used to render text data, either with char or wchar type codes. The resource factory allows you to declare a specific code set identifier used natively for either char or wchar data. The native code set is what is used when interacting with users, files, printers, etc.

Occasionally, it is necessary to interoperate with other processes using different native code sets. For each peer to render the data correctly, the character data must be transformed from one code set to another. This transformation is the responsibility of a code set translator. The resource factory is responsible for managing code set translators for char and wchar data. The translators themselves are created by factory objects, which are separately-loaded service objects. It is possible to configure many translator factories, but only the translators that correspond to the configured native char or wchar code set will participate in inter-ORB communication.

Four resource factory options, shown in table Table 20-9, control the configuration of code set information.

Table 20-8 Options for Adding IOR Parsers

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBIORParser parser_name</td>
<td>20.6.7</td>
<td>Specify the service name of an IOR parser that should be added to the resource factory.</td>
</tr>
</tbody>
</table>
When processing long duration requests, such as a database access or media I/O, a service can take advantage of *asynchronous method handling* (AMH) to decouple the receipt of a request from the processing of that request. AMH allows a request to be received by one thread, processed by another, and possibly have the reply issued by a third. The server’s ORB creates an AMH response handler when the request is received. When request processing is completed, this response handler is used to generate the reply sent back to the client. For a detailed discussion of AMH, see Chapter 8.

Similarly, clients can use *asynchronous method invocation* (AMI) to avoid blocking while waiting for a reply. The client’s ORB creates an AMI response handler whenever an operation is invoked asynchronously on a remote proxy. The response handler asynchronously receives the reply from the server and delivers it to the proxy. For a detailed discussion of AMI, see Chapter 7.

The resource factory can be configured to use different types of allocators to create AMH and AMI response handlers. The resource factory interface provides the following operations that are used by the ORB to access the allocators used to create AMH and AMI response handlers:

```c
virtual ACE_Allocator* amh_response_handler_allocator (void);
virtual ACE_Allocator* ami_response_handler_allocator (void);
```

### Table 20-9 Options for Configuring Code Sets

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBNativeCharCodeset id</td>
<td>20.6.9</td>
<td>Specify the native code set identifier for char data. The identifier may be expressed by either a number or a locale name.</td>
</tr>
<tr>
<td>-ORBNativeWCharCodeset id</td>
<td>20.6.10</td>
<td>Specify the native code set identifier for wchar data. The identifier may be expressed by either a number or a locale name.</td>
</tr>
<tr>
<td>-ORBCharCodesetTranslator factory</td>
<td>20.6.1</td>
<td>Name a translator available to convert between the native char code set and some other code set.</td>
</tr>
<tr>
<td>-ORBWCharCodesetTranslator factory</td>
<td>20.6.15</td>
<td>Name a translator available to convert between the native wchar code set and some other code set.</td>
</tr>
</tbody>
</table>
In a multithreaded environment, a locked allocator may be required. In a resource constrained environment, such as an embedded system where memory usage is a concern, the creation of locks may be undesirable. In such cases, it is possible to configure the resource factory to create lock-free allocators. By default, the advanced resource factory will create locked allocators.

The advanced resource factory provides options to configure the type of allocators used for AMH and AMI response handlers. These options are shown in Table 20-10.

### Table 20-10 Options for Configuring AMH and AMI Response Handler Allocators

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBAMHResponseHandlerAllocator {thread</td>
<td>20.7.1</td>
<td>Specify the type of locking used when accessing the AMH response handler allocator.</td>
</tr>
<tr>
<td></td>
<td>null}</td>
<td></td>
</tr>
<tr>
<td>-ORBAMIResponseHandlerAllocator {thread</td>
<td>20.7.2</td>
<td>Specify the type of locking used when accessing the AMI response handler allocator.</td>
</tr>
<tr>
<td></td>
<td>null}</td>
<td></td>
</tr>
</tbody>
</table>

#### 20.2.10 CORBA Object Synchronization

Locking is needed to synchronize access to the internal state of proxies to remote CORBA objects when those proxies are shared by multiple threads. An example of such internal state is the reference counts used to manage the memory associated with the proxies. Single-threaded clients do not need locking to protect the internal state and can therefore reduce memory usage and locking overhead by using a null locking strategy.

The resource factory interface defines the following operation, which returns the type of lock that should be used when threads access proxies to remote CORBA objects.

```c
virtual ACE_Lock* create_corba_object_lock (void);
```

The possible lock types returned from this operation are a null lock and a thread-safe lock. The `null` strategy provides no locking to synchronize access to the internal state of proxies to remote CORBA objects. The `thread` strategy provides thread-safe access to this state. The default resource factory returns a thread-safe lock by default.
Table 20-11 shows the option to configure the locking strategy used by the ORB to synchronize access to the internal state of proxies to remote CORBA objects.

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`-ORBCorbaObjectLock</td>
<td>20.6.5</td>
<td>Selects the type of lock threads use to access the internal state of proxies to remote CORBA objects. The default behavior is to use a thread-safe locking strategy.</td>
</tr>
</tbody>
</table>

### 20.2.11 Resource Usage Strategy

The ORB is responsible for creating various resources dynamically. For example, when an object reference is received as part of a request or is returned in a reply, the IOR is encoded in a input CDR stream. The ORB must extract the IOR from the input CDR stream before the application can use it. The ORB can either create a stub immediately upon extracting the IOR, which is computationally expensive and resource intensive, or it can create a simple encapsulation of the IOR and defer stub creation until a stub is actually needed by the application (for example, to invoke a request via the object reference).

Some applications, such as those that deal with hundreds or even thousands of object references (e.g., the Naming Service), can benefit by deferring stub creation if a stub is not needed immediately. The benefits are realized in terms of run time memory footprint and performance when large numbers of object references are transmitted.

The ORB uses a resource usage strategy to determine whether it should create such resources immediately or defer creating them until needed by the application. The following operation of the resource factory interface allows the ORB to access the resource usage strategy:

```c++
enum Resource_Usage{
   TAO_EAGER,
   TAO_LAZY
};

virtual TAO_Resource_Factory::Resource_Usage resource_usage_strategy (void)
const = 0;
```
The `resource_usage_strategy()` operation returns the type of resource usage strategy to be used by the ORB. There are two defined resource usage strategies that may be returned from this operation:

- *eager* (TAO_EAGER)—this strategy instructs the ORB to create certain resources immediately.
- *lazy* (TAO_LAZY)—this strategy instructs the ORB to defer creation of certain resources.

**Note** Currently, the resource usage strategy only affects the creation of stubs when object references are extracted from input CDR streams. If the eager strategy is used, a stub is created for each object reference as it is extracted. If the lazy strategy is used, a simple encapsulation of the object reference is created and a stub is only created if the application later uses the reference. The behavior of the resource usage strategy may be extended in the future to affect the creation of other types of resources, as well.

By default, the default resource factory provides an eager resource usage strategy. This default behavior can be changed at compile time by defining the following preprocessor macro in your `$ACE_ROOT/ace/config.h` file:

```
#define TAO_USE_LAZY_RESOURCE_USAGE_STRATEGY 1
```

The type of resource usage strategy can also be changed at run time by using the `-ORBResourceUsage` resource factory option as shown in Table 20-12.

**Table 20-12 Resource Usage Strategy Configuration**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ORBResourceUsage</code></td>
<td>20.6.14</td>
<td>Determines how the ORB uses resources when creating object references. The default is <code>eager</code>.</td>
</tr>
</tbody>
</table>

### 20.3 Resource Factory for Qt GUI Toolkit

If you are building a TAO application that is integrated with the Qt GUI toolkit from Trolltech [http://www.trolltech.com], you must use the ACE_QtReactor. For example, a TAO CORBA server that also has a graphical user interface, developed with the Qt toolkit, needs to be responsive...
to both CORBA requests and Qt events. Such an application must use the
ACE_QtReactor.

**Note**  
To use the ACE_QtReactor, you must place `#define ACE_HAS_QT` in your
$ACE_ROOT/ace/config.h file when you build ACE and TAO.

Initializing the ACE_QtReactor requires supplying a valid QApplication
pointer to the reactor constructor. A specialized resource factory is supplied
with TAO, providing the means for integrating with the Qt toolkit. This
factory, called the TAO_QtResource_Factory, is derived from the default
resource factory and may be used in its place.

The TAO_QtResource_Factory accepts the same options as the default
resource factory. To use the TAO_QtResource_Factory, you must add a
directive to your service configuration file similar to the directive shown here:

```
dynamic Resource_Factory Service_Object* TAO:_make_TAO_QtResource_Factory()"
```

In addition to loading the new resource factory, a valid QApplication
pointer must be supplied to the factory before TAO attempts to initialize the
reactor. The file `$TAO_ROOT/tests/QtTests/client.cpp` contains the
following example code:

```cpp
#include "testC.h"
#include <ace/Get_Opt.h>
#include "client.h"

int main (int argc, char* argv[])
{
    QApplication app (argc, argv);
    TAO_QtResource_Factory::set_context (&app);
    TAO_ENV_DECLARE_NEW_ENV;
    ACE_TRY
    {
        CORBA::ORB_var orb =
            CORBA::ORB_init (argc, argv, "" TAO_ENV_ARG_PARAMETER);
        ACE_TRY_CHECK;
        //...
    }
    //...
    ACE_ENDTRY;
    return 0;
}
```
20.4 Resource Factory for X Windowing Toolkit

In this example, we create a `QApplication` object and supply its address to the `TAO_QtResource_Factory` via the static `set_context()` function. Once the factory is initialized, the application proceeds to initialize and use the ORB as usual.

20.4 Resource Factory for X Windowing Toolkit

If you are building a TAO application that is integrated with the X Window System’s Xt Intrinsics toolkit, you must use the `ACE_XtReactor`. For example, a TAO CORBA server that also has a graphical user interface, developed with an Xt-based toolkit, needs to be responsive to both CORBA requests and Xt events. Such an application must use the `ACE_XtReactor`.

Note

To use the `ACE_XtReactor`, you must place `#define ACE_HAS_XT` in your `$ACE_ROOT/ace/config.h` file when you build `ACE` and `TAO`.

Initializing the `ACE_XtReactor` requires supplying an `XtAppContext` object to the reactor constructor. A specialized resource factory is supplied with TAO, providing the means for integrating with the Xt toolkit. This factory, called the `TAO_XT_Resource_Factory`, is derived from the default resource factory and may be used in its place.

The `TAO_XT_Resource_Factory` accepts the same options as the default resource factory. To use the `TAO_XT_Resource_Factory`, you must add a directive to your service configuration file similar to the directive shown here:

```
dynamic Resource.Factory Service_Object* TAO: make_TAO_XT_Resource_Factory()"
```

In addition to loading the new resource factory, an `XtAppContext` must be supplied to the factory before TAO attempts to initialize the reactor. The file `$TAO_ROOT/tests/Xt_Stopwatch/client.cpp` contains the following example code:

```
#include "testC.h"
#include <ace/Get_Opt.h>
#include "Control.h"
#include "Client.h"
```
int main (int argc, char* argv[])
{
    XtAppContext  app;
    Widget        toplevel = XtAppInitialize (&app, "Start & Stop", NULL, 0,
                                      &argc, argv, NULL, NULL, 0);
    TAO_XT_Resource_Factory::set_context (app);
    // ...
    ACE_DECLARE_NEW_CORBA_ENV;
    ACE_TRY
    {
        CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "", TAO_ENV_ARG_PARAMETER);
        ACE_TRY_CHECK;
        // ...
    }
    // ...
    ACE_ENDTRY;
    return 0;
}

In this example, we create an XtAppContext object, initialize the top level
widget, and supply the context object to the TAO_XT_Resource_Factory
via the static set_context() function. Once the factory is initialized, the
application proceeds to initialize and use the ORB as usual.

## 20.5 Advanced Resource Factory

The advanced resource factory gives more control than the default resource
factory over the types of resources used and how those resources are accessed.
In addition to the options provided by the default resource factory, the
advanced resource factory provides options that allow selecting different
reactors, changing the locking method on input CDR allocators, and adjusting
the connection caching strategy. The advanced resource factory was created to
allow more advanced options, while keeping the footprint of the default
resource factory small.

The advanced resource factory inherits from the default resource factory and
accepts all of that factory’s options, in addition to its own. It can be loaded
dynamically using a service configurator directive of the form shown below,
all of which should be on one line:

```bash
dynamic Advanced_Resource_Factory Service_Object* 
TAO_Strategies: make_TAO_Advanced_Resource_Factory () "-ORBoption value 
-ORBoption value ..."
```
It can also be loaded statically as follows:

- Add the following preprocessor `#include` directive to the file containing `main()`:

  ```
  #include "tao/Strategies/advanced_resource.h"
  ```

  You can omit this header file if you always use dynamic libraries.

- Link the `TAO_Strategies` library into the executable, or just inherit your MPC project from the `strategies` base project.

- Specify a service configurator directive of the form:

  ```
  static Advanced_Resource_Factory "-ORBoption value -ORBoption value ..."
  ```

  See 18.3 for more information on using the service configurator. Once you have loaded the advanced resource factory, directives for the default resource factory (i.e., `Resource_Factory`) have no effect and will generate warnings.

One of the key features of the advanced resource factory is the ability to select a different reactor for use with the ORB. This variability enables the use of TAO in an ACE-based application that needs a specialized reactor for different types of events. For instance, the `Fl_Reactor` is used to integrate with applications that receive events from the “fast lite” windowing system, and the `msgWFMOS_Reactor` is used with Win32-based applications that support the COM messaging model as well.

The Qt and Xt reactors are not selected using the advanced resource factory. These reactors require specialized resource factories because the correct application contexts must be supplied to the reactor prior to ORB initialization. The Qt resource factory (see 20.3) and Xt resource factory (see 20.4) are specializations of the default resource factory, so they can provide this capability.

For additional information on the different types of reactors and their uses, please refer to the appropriate ACE documentation via <http://www.theaceorb.com/references/>.
The additional options provided by the advanced resource factory are summarized in Table 20-13.

### Table 20-13 Additional Options provided by Advanced Resource Factory

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBAMHResponseHandlerAllocator thread</td>
<td>20.7.1</td>
<td>Specify the type of locking used when accessing the AMH response handler allocator.</td>
</tr>
<tr>
<td>-ORBAMIResponseHandlerAllocator thread</td>
<td>20.7.2</td>
<td>Specify the type of locking used when accessing the AMI response handler allocator.</td>
</tr>
<tr>
<td>-ORBCConnectionPurgingStrategy lru</td>
<td>20.7.3</td>
<td>Select the strategy for determining the connections to purge from the cache.</td>
</tr>
<tr>
<td>-ORBInputCDRAllocator thread</td>
<td>20.7.4</td>
<td>Specify the type of locking used when accessing the input CDR allocator.</td>
</tr>
<tr>
<td>-ORBReactorThreadQueue lifo</td>
<td>20.7.5</td>
<td>Choose the strategy for determining the order in which waiting threads are selected to run the reactor’s event loop when using the thread-pool reactor type.</td>
</tr>
<tr>
<td>-ORBReactorType fl msg wfmo</td>
<td>20.7.6</td>
<td>Use this option to select an alternative type of reactor for use with the ORB.</td>
</tr>
</tbody>
</table>
20.6 Resource Factory Options

This section describes the individual options interpreted by the default resource factory and any of its specializations, such as the Qt and Xt resource factories. These options are supplied to the resource factory by the service configurator through the use of a static initialization directive (see 18.3).

This section does not describe the additional options supported by the advanced resource factory. See 20.7 for those options.
20.6.1 ORBCharCodesetTranslator factory

Description
Character data may require conversion from one code set to another in order to interoperate with applications using different native code sets. While the translator factory objects are loaded by their own service configuration directive, this option causes the resource factory to evaluate the specified factory to see if its translator is compatible with this application’s native code set.

Usage
There is no default code-set translator defined. Translators are declared by the name given to the factory object that loads it. TAO may be used successfully without loading any translators. In this case, the process is simply limited to communicating in its configured native char code set. Many translators may be configured, by repeating this option for each translator.

Impact
Each char code set translator works by converting text data sent to a remote process from the native char code set to a mutually agreed upon transmission code set. The translator similarly converts text data received from a remote process into the native char code set from the transmission code set.

Having many translators present increases the ORB’s ability to communicate with a wide variety of remote processes. The code set used in transmission may be the native code set for either side, or may be a third code set. If it is the native code set for both sides, no translation is required. Otherwise, the non-native side will have to translate the data. It is possible to use a third code set that is translated by both the sides of the connection.

See Also
20.2.8, 20.6.9

Example
dynamic AsciiToEbcdic Service_Object * mycodesetlib: make_AsciiToEbcdic ()
static Resource_Factory "-ORBCharCodesetTranslator AsciiToEbcdic"
20.6 Resource Factory Options

20.6.2 ORBConnectionCacheLock  \textit{lock\_type}

\textbf{Values for lock\_type}

<table>
<thead>
<tr>
<th>lock_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread (default)</td>
<td>Specifies inter-thread mutex to guarantee exclusive access.</td>
</tr>
<tr>
<td>null</td>
<td>No locking.</td>
</tr>
</tbody>
</table>

\textbf{Description} TAO uses a cached-connection strategy to improve efficiency in establishing connections. This strategy allows for the reuse of connection handlers, if possible, rather than creating new ones.

In a multithreaded environment, access to the cache must be synchronized. This option specifies the type of lock used to protect the connection cache.

\textbf{Usage} The default is acceptable for use in all applications. In applications that are single-threaded, or those for which all interaction with a connection is through a single thread in a multithreaded application, a null lock may be used.

A null lock must not be used when multiple threads are obtaining connections from the cache.

\textbf{Impact} Use of a null lock will reduce the latency involved with obtaining a connection. The null mutex class, supplied by ACE to minimize code changes when switching from multi- to single-threaded applications, reduces all locking operations to no-ops.

\textbf{Note} \textit{When using the advanced resource factory, it is possible for this option and the -ORBInputCDRAlocator option to have conflicting values. Both options affect data-block locking. If conflicting values are given, the value provided by -ORBConnectionCacheLock will be used. If you use both of these options, they should have the same setting to prevent unexpected results.}

\textbf{See Also} 20.2.2, 20.7.4

\textbf{Example} static Resource\_Factory "-ORBConnectionCacheLock null"
20.6.3 **ORBConnectionCacheMax limit**

**Description**  
Opened connections are added to the cache to facilitate their reuse. If a process continues to run, and these connections are not reused appropriately, the cache will continue to grow. Therefore, before each new connection, the cache is checked and purged if it has reached the limit specified by this option. The default is system dependent, but can be overridden at compile time by defining the preprocessor macro `TAO_CONNECTION_CACHE_MAXIMUM`.

**Usage**  
Determining an appropriate limit for the cache may be completely dependent upon available resources.

**Impact**  
The larger the cache limit, the more connection resources that may be used.

**See Also**  
20.6.4

**Example**  
```
static Resource_Factory "-ORBConnectionCacheMax 1024"
```
20.6.4 ORBConnectionCachePurgePercentage percent

Description
Opened connections are added to the cache to facilitate their reuse. If a process continues to run, and these connections are not reused appropriately, the cache will continue to grow. Therefore, before each new connection, the cache is checked and purged if it has reached the limit specified by the -ORBConnectionCacheMax option or the system default, if that option was not used. This option is used to set the amount of the connection cache actually purged when it is time to do so. The default amount is twenty (20) percent. If zero is supplied as a percent, then no cache purging will occur, regardless of the strategy selected. A value less than zero will be interpreted as zero and a value 100 or greater will result in removal of all cache entries.

Usage
Determining an appropriate percentage of the cache to remove at one time requires balancing the time required to actually perform the purge versus how often purging will be necessary.

Impact
The act of purging connections takes time. Depending on the frequency of connection creation, a large percentage may be necessary to lower the time between purges.

See Also
20.6.3

Example
static Resource_Factory "-ORBConnectionCachePurgePercentage 50"
20.6.5  ORBCorbaObjectLock  lock_type

Values for lock_type

<table>
<thead>
<tr>
<th>lock_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>The internal state (e.g., reference count) of CORBA object proxies is protected using a thread-safe locking strategy.</td>
</tr>
<tr>
<td>null</td>
<td>No locking is used to protect the internal state of CORBA object proxies.</td>
</tr>
</tbody>
</table>

Description  When proxies to remote CORBA objects are shared among multiple threads, access to their internal state (such as their reference counts) must be synchronized. The -ORBCorbaObjectLock resource factory option allows the user to specify the locking strategy to be used when accessing and manipulating the internal state of remote CORBA object proxies.

Usage  The default value of thread is appropriate for most applications. A thread-safe locking strategy allows multiple threads to safely use and manipulate proxies to remote CORBA objects.

Use the null strategy to reduce the memory and performance overhead associated with locks. The null strategy should only be used in single-threaded applications.

See Also  20.2.10

Example  static Resource.Factory "-ORBCorbaObjectLock null"
20.6.6 ORBFlushingStrategy strategy

Values for strategy

<table>
<thead>
<tr>
<th>strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader_follower (default)</td>
<td>Use the reactor and non-blocking I/O to send the outgoing messages. This strategy participates in the leader-follower pattern to synchronize access to the reactor. Use the reactor, but do not take part in the leader-follower pattern. This strategy is better used only in single-threaded applications. Use the blocking strategy, which flushes the queue as soon as it becomes “full,” and blocks the thread until all the data is sent.</td>
</tr>
</tbody>
</table>

Description

When one ORB must communicate with another, it looks at the request, creates a message that will be understood by the other ORB, then sends the message. The parts of the message are stored as ACE_Message_Block objects. The ACE_Message_Block objects build up in a queue before they are flushed and the data in them is sent. You can adjust the strategy by which the messages are flushed using this option. For example, you can use this option to alter the timing of message sending in your application.

Usage

The default value is appropriate for most applications. A slight performance gain can be achieved in a single-threaded application when the reactive strategy is used because it does not perform any locking to synchronize access to the reactor. Use the blocking strategy to avoid allowing the thread that is waiting to flush messages to be made available to the ORB for potentially handling incoming requests.

Impact

The only optimization benefit is gained in a single-threaded application. In this configuration, the reactive strategy will remove synchronization that controls access to the reactor when sending messages. The blocking strategy ensures that the thread that is waiting to write to the transport will block until it can write and cannot be used by the ORB to dispatch incoming requests. The blocking strategy can lead to poor responsiveness and potential deadlocks if all the threads used by the ORB are blocked waiting to write and none is available to respond to incoming requests.

See Also

17.4.3

Example

static Resource_FACTORY "-ORBFlushingStrategy reactive"
20.6.7 ORBIORParser parser_name

**Description**
TAO supports several IOR formats, including corbaloc:, corbaname:, IOR:, and file:. However, some applications may benefit from other formats. For example, http: could allow applications to download an object reference from a web server. This option allows application developers to implement their own IOR parsers and dynamically add them to the ORB.

Using an IOR parser is more convenient than adding configuration code in the application’s main() function. It also allows for easier integration with other TAO components, such as the -ORBInitRef options.

**Usage**
First create an IOR parser class that inherits from TAO_IOR_Parser. The `parser_name` should be the ACE Service Configurator service name you defined using the `ACE_STATIC_SVC_DEFINE` macro in your IOR Parser class. Your custom parser should not attempt to match one of the predefined IOR prefixes. Redefining or matching one of the predefined IOR prefixes may cause unexpected behavior.

You can repeat this option to add multiple parsers.

**Impact**
Additional IOR parsers can impact performance when calling the `string_to_object()` operation. Each IOR parser added is queried to see if it is a match. If it is a match, the parser is used to return the object reference. The additional match checking and any overhead within the custom parser’s code to look up the object reference can impact performance.

**See Also**
20.2.7

**Example**
static Resource_Factory "-ORBIORParser HTTP_Parser"
20.6.8 ORBMuxedConnectionMax *limit*

**Description**
Opened connections are added to the cache to facilitate their reuse. Multiple connections may be established for a given remote endpoint and stored in the cache. This option allows an application to limit the number of connections that can be established for a given remote endpoint.

**Usage**
Determining an appropriate limit for the number of connections that can be established for a given remote endpoint may be completely dependent upon available resource and the number of client threads that are likely to be concurrently invoking requests on objects in a given remote server.

**Impact**
The larger the limit, the more per-endpoint connections that may be established.

**See Also**
20.2.4, 22.3.3

**Example**
static Resource_Factory "-ORBMuxedConnectionMax 1"
Resource Factory

20.6.9  ORBNativeCharCodeset id

Values for id

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0c00010001 (default)</td>
<td>The Code Set Registry id for ISO 8859-1, the CORBA defined default for 8-bit character codes.</td>
</tr>
<tr>
<td>some other numeric code or Local Name</td>
<td>When not using the default code set, use some other Code Set Registry Id value or locale name.</td>
</tr>
</tbody>
</table>

Description

The native code set for characters is described by an entry in the Open Software Foundation’s (OSF) Code and Character Set Registry, currently version 1.2g. This value is embedded in object-reference profiles as a declaration of the code set used to render text data written to files, consoles, printers, etc. By default, TAO uses ISO 8859-1 as its character code set.

Usage

Use this option to configure a different code set for character data. This may be useful if you are deploying an application internationally, where different installations may render text differently.

In order to use locale names in place of numeric values, you must separately configure the code-set registry database in ACE to assign locale names to the individual entries. The OSF does not define locale names for entries. The code set registry database shipped with TAO defines entries for “ASCII,” and “EBCDIC” as locale names for two common 8-bit code sets.

It is also possible to compile with an alternative code set id for char data by changing the value of TAO_DEFAULT_CHAR_CODESET_ID in the source file $TAO_ROOT/tao/corbafwd.h. Of course, this change requires the TAO library be rebuilt, and affects every application that links to it. A default code set id changed this way may still be overridden by the resource factory.

Impact

Altering the native code set impacts information used to define the capabilities of an ORB process. When then native code set of a client differs from that of a server, the client must chose an alternate code set based on any translators in either the client or server. If no suitable translator is available, then a client will not be able to exchange text data with a server.

See Also

20.2.8, 20.6.1

Example

static Resource_Factory "-ORBNativeCharCodeset 0x10020417"
20.6 Resource Factory Options

20.6.10 ORBNativeWCharCodeset \textit{id}

Values for \textit{id}

<table>
<thead>
<tr>
<th>\textit{id}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00010109 (default)</td>
<td>The Code Set Registry id for UTF-16, also known as unicode.</td>
</tr>
<tr>
<td>some other numeric code or Local Name</td>
<td>When not using the default code set, use some other Code Set Registry Id value or locale name</td>
</tr>
</tbody>
</table>

Description

The native code set for wide characters is described by an entry in the Open Software Foundation’s (OSF) Code and Character Set Registry, currently version 1.2g. This value is embedded in object reference profiles as a declaration of the code set used to render text data written to files, consoles, printers, etc. By default, TAO uses UTF-16 as its wide character code set. This default is a 2-octet code set commonly used by MS Windows applications. On platforms such as Solaris, UCS-4, a 4-octet code set, is a better choice.

Usage

Use this option to configure a different code set for wide character data. This may be useful if you are deploying an application internationally, where different installations may render text differently.

In order to use locale names in place of numeric values, you must separately configure the code set registry database in ACE to assign locale names to the individual entries. The OSF does not define locale names for entries. The code set registry database shipped with TAO defines entries for “UCS-4,” and “Unicode” as locale names for two common wide character code sets.

It is also possible to compile with an alternative code set id for char data by changing the value of \texttt{TAO_DEFAULT_WCHAR_CODESET_ID} in the source file $\$TAO_ROOT/tao/corbafwd.h. Of course, this change requires the TAO library be rebuilt, and affects every application that links to it. A default code set id changed this way may still be overridden by the resource factory.

Impact

Altering the native code set impacts information used to define the capabilities of an ORB process. When then native code set of a client differs from that of a server, the client must chose an alternate code set based on any translators in either the client or server. If no suitable translator is available, then a client will not be able to exchange text data with a server.

See Also

20.2.8, 20.6.15

Example

\texttt{static Resource\_Factory "-ORBNativeWCharCodeset 0x00010104"}
## 20.6.11 ORBOBJECTKeYTableLock *lock_type*

### Values for *lock_type*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread (default)</td>
<td>The object key table should be protected by a thread-safe lock.</td>
</tr>
<tr>
<td>null</td>
<td>No locking is performed when accessing the object key table.</td>
</tr>
</tbody>
</table>

### Description

Many object references may share an object key. Rather than repeating this information in every IOR, a table is used to store object keys. The object key table keeps track of the object keys that are generated and made available through IORs. The table manages the lifespan of the object keys within the ORB through reference counting.

This option is used to specify the type of lock to be used when accessing the object key table. Possible values for lock type are `thread`, which specifies that a thread-safe mutex is used to guarantee exclusive access, and `null`, which specifies that no locking is to be performed. The default value is `thread`.

### Usage

The default value for this option is suitable for most applications and should be used by all multithreaded applications. Single-threaded applications can use the `null` value for this option to avoid the memory consumption for the lock and the locking overhead when the object key table is accessed.

Both server and client applications may use this option.

### See Also

20.2.5

### Example

```bash
static Resource_Factory "-ORBOBJECTKeYTableLock null"
```
20.6.12 ORBProtocolFactory *factory_name*

Values for *factory_name*

<table>
<thead>
<tr>
<th><em>factory_name</em></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIOP.Factory (default)</td>
<td>The factory providing protocol elements to support Internet Inter-ORB Protocol.</td>
</tr>
<tr>
<td>DIOP.Factory</td>
<td>Support for GIOP over UDP/IP. This is not interoperable with other ORBs.</td>
</tr>
<tr>
<td>IIOP_Lite.Factory</td>
<td>Support for IIOP using GIOP-Lite. This is not interoperable with other ORBs.</td>
</tr>
<tr>
<td>SHMIOP.Factory</td>
<td>Support for Shared Memory Inter-ORB Protocol. This is not interoperable with other ORBs.</td>
</tr>
<tr>
<td>SSLIOP.Factory</td>
<td>Support for Secure Socket Layer Inter-ORB Protocol.</td>
</tr>
<tr>
<td>UIOP.Factory</td>
<td>Support for Unix domain (local IPC) Inter-ORB Protocol. This is not interoperable with other ORBs.</td>
</tr>
<tr>
<td>UIOP_Lite.Factory</td>
<td>Support for UIOP using GIOP-Lite. This is not interoperable with other ORBs.</td>
</tr>
<tr>
<td>custom factory</td>
<td>Application developers may produce specialized protocol factories that satisfy the requirements of the interface and provide behavior not already supplied with TAO.</td>
</tr>
</tbody>
</table>

**Description**

This option is used to “plug” pluggable protocols into TAO. Pluggable protocols offer application designers the ability to deploy TAO-based objects in an environment other than one using TCP/IP. The protocols specified here are the transport layer, not the messaging layer. The inter-ORB messaging protocol is GIOP. Pluggable protocols are discussed in depth in Chapter 15 and Chapter 16.

A GIOP message consists of a message header and, for request messages, a sub-header containing additional information. The GIOP version 1.2 message header contains the following fields: GIOP magic number; GIOP version; GIOP flags; message type; and message length. For request messages there is an additional request header that contains the following fields: service context; request identifier; response flags; target address; and operation. Applications operating in homogeneous (e.g., embedded, real-time) environments do not require all of these fields.

For these environments, the TAO-specific lightweight version of GIOP (GIOP-Lite), which is used by IIOP-Lite and UIOP-Lite, removes the *magic number*, *version*, and *byte-order flag* fields from the GIOP header and the *service context* field from the GIOP request header. The resulting savings of
several bytes per request can improve performance. Message transfer
overhead and thus latencies are also reduced.

Since GIOP-Lite removes fields from GIOP messages, IIOP-Lite and
UIOP-Lite can only be used in homogeneous environments (e.g., where all the
connected platforms have the same byte order and where TAO is the only
ORB). If one of these is used, it must be used for all connected processes.
Results are unpredictable if client and server are not both using the Lite
version of the protocol.

See 15.11 and 29.9 for more information on using the SSLIOP_Factory and
using SSL with TAO.

Usage

Multiple protocol factories may be added to the resource factory. Every time
the option is repeated, the specified protocol factory is added to a list of
available protocol factories.

The default, IIOP_Factory, must be specified if that protocol is desired,
along with any other protocol. If UIOP is available but IIOP is desired
exclusively, then IIOP_Factory must be specified as the only protocol
factory. If one of the Lite protocols is used, it must be used for all connected
processes.

Note

If one of the Lite protocols is used, it must be used for all connected processes.
Additionally, the UIOP-Lite protocol requires that the client and server be run
on the same host. UIOP-Lite also requires host operating system support of
local IPC.

Impact

Additional protocols impact performance at the initialization of servers. For
each protocol initialized, a profile is added to an IOR, increasing its size.

See Also

19.8.10

Example
dynamic IIOP_Lite_Factory Service_Object *
    TAO: make_TAO_IIOP_Lite_Protocol_Factory() ""

dynamic My_Protocol_Factory Service_Object * mylib: make_my_protocol_factory()
    ""

static Resource_Factory "-ORBProtocolFactory IIOP_Lite_Factory \ 
    -ORBProtocolFactory UIOP_Factory \ 
    -ORBProtocolFactory My_Protocol_Factory"
20.6.13 ORBReactorMaskSignals state

Values for state

<table>
<thead>
<tr>
<th>state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Do not mask signals.</td>
</tr>
<tr>
<td>1 (default)</td>
<td>Mask signals.</td>
</tr>
</tbody>
</table>

Description
When using a select-based reactor, this option provides an additional level of control. The ACE_Select_Reactor, in any of its threaded variations, ordinarily masks signals during its operation. This provides a level of signal-safety during these operations. Signal masking involves calls to the kernel to mask the signals, then later to unmask them. The constructor for ACE_Select_Reactor accepts a value to control this behavior. The default for this value is 1, meaning “yes, mask signals,” but may also be 0 meaning “no, do not mask signals.” The option -ORBReactorMaskSignals provides the TAO developer the means for controlling this behavior.

Usage
If your application registers signal handlers with the reactor, the default value for this option is appropriate. Applications that do not use signal handlers and wish to have the greatest performance possible may see a slight performance gain by not masking signals.

Impact
This option controls the behavior of the ACE_Select_Reactor, and only on non-win32 platforms. Turning off signal masking in applications that also register signal handlers with the reactor may cause unexpected behavior.

See Also
20.7.6

Example
static Resource_Factory "-ORBReactorMaskSignals 0"
## 20.6.14 ORBResourceUsage usage_type

**Values for usage_type**

<table>
<thead>
<tr>
<th>usage_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eager (default)</td>
<td>The eager strategy instructs the ORB to create certain types of resources (e.g., stubs) immediately.</td>
</tr>
<tr>
<td>lazy</td>
<td>The lazy strategy instructs the ORB to defer creation of certain types of resources until actually needed by the application.</td>
</tr>
</tbody>
</table>

### Description

This option allows you to control the type of resource usage strategy used by the ORB. The ORB uses the resource usage strategy to determine whether it should create certain types of resources immediately or defer creating them until needed by the application.

By default, an eager resource usage strategy is used. This strategy assumes the application will immediately want to use resources, such as object references. The lazy resource usage strategy allows the ORB to defer creation of such resources until the application attempts to use them, if at all.

### Usage

Currently, this option only affects the creation of stubs when object references are extracted from input CDR streams. With the eager resource usage strategy, a stub is immediately created as each object reference is extracted. With the lazy resource usage strategy, a stub is not created immediately. Instead, a simple encapsulation of the object reference is created upon extraction and actual stub creation is deferred until the application actually attempts to use the object reference (e.g., to invoke a request).

Some applications, such as the Naming Service, deal with hundreds or even thousands of object references, but may never use them to invoke requests. Such applications can benefit from the lazy resource usage strategy to avoid creating unnecessary stubs. The benefits are realized in terms of reduced memory footprint and better performance when object references are transmitted.

The default resource usage strategy can be changed by adding the following preprocessor macro definition to your $ACE_ROOT/ace/config.h file and recompiling TAO:

```c
#define TAO_USE_LAZY_RESOURCE_USAGE_STRATEGY 1
```

### See Also

20.2.11

### Example

```
static Resource_Factory "-ORBResourceUsage lazy"
```
20.6.15 ORBWCharCodesetTranslator factory

Description
Wide character data may require conversion from one code set to another in order to interoperate with applications using different native code sets. While the translator factory objects are loaded by their own service configuration directive, this option causes the resource factory to evaluate the specified factory to see if its translator is compatible with this application’s native code set.

Usage
There is no default code set translator defined. Translators are declared by the name given to the factory object that loads it. TAO may be used successfully without loading any translators. In this case the process is simply limited to communicating in its configured native char code set. Many translators may be configured, by repeating this option for each translator.

Impact
Each wchar code set translator works by converting text data sent to a remote process from the native wchar code set to a mutually-agreed-upon transmission code set. The translator similarly converts text data received from a remote process into the native wchar code set from the transmission code set.

The number of octets transmitted per character depends on the maximum character size as defined by the code set specification, not the width of a wchar in the native environment. For instance, if a host platform defines wide characters as being 4 octets, but the selected code set has a maximum width of 2 octets, then only 2 octets will be transmitted.

Having many translators present increases the ORB’s ability to communicate with a wide variety of remote processes. The code set used in transmission may be the native code set for either side, or may be a third code set. If it is the native code set for both sides, no translation is required. Otherwise, the non-native side will have to translate the data. It is possible to use a third code set that is translated by both sides of the connection.

See Also
20.2.8, 20.6.10

Example
The following example shows how to load a hypothetical wchar code set translator for converting between the UCS-4 (32-bit) code set and the UTF-16 (16-bit) code set. The name given to our translator is UCS4ToUTF16, and it is loaded from the MyTranslators library.

dynamic UCS4ToUTF16 Service Object* MyTranslators::make_UCS4ToUTF16()
static Resource_Factory "-ORBWCharCodesetTranslator UCS4ToUTF16"
20.7 Advanced Resource Factory Options

This section describes the individual options interpreted by the advanced resource factory. The advanced resource factory supports these options in addition to those supported by the default resource factory described in 20.6. See 20.5 for more information on using the advanced resource factory.
## 20.7.1 ORBAMHResponseHandlerAllocator type

### Values for type

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>thread</code> (default)</td>
<td>Uses a thread-safe allocator.</td>
</tr>
<tr>
<td><code>null</code></td>
<td>Uses non thread-safe allocator.</td>
</tr>
</tbody>
</table>

### Description
Specify the type of locking used when accessing the AMH response handler allocator.

### Usage
The default value of `thread` is appropriate for most applications. A thread-safe AMH response handler allocator allows multiple threads to safely access the allocator.

Use the `null` strategy to reduce the overhead associated with locking for single-threaded applications.

This option only affects servers that use AMH.

### See Also
.20.2.9, Chapter 8

### Example
```bash
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies: make_TAO_Advanced_Resource_Factory ()
"-ORBAMHResponseHandlerAllocator null"
```
20.7.2 ORBAMIResponseHandlerAllocator type

Values for which

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread (default)</td>
<td>Uses a thread-safe allocator.</td>
</tr>
<tr>
<td>null</td>
<td>Uses non thread-safe allocator.</td>
</tr>
</tbody>
</table>

Description
Specify the type of locking used when accessing the AMI response handler allocator.

Usage
The default value of thread is appropriate for most applications. A thread-safe AMI response handler allocator allows multiple threads to safely access the allocator.

Use the null strategy to reduce the overhead associated with locking for single-threaded applications.

This option only affects clients that use AMI.

See Also
.20.2.9, Chapter 7

Example
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies::make_TAO_Advanced_Resource_Factory ()
"-ORBAMIResponseHandlerAllocator thread"
20.7.3 ORBConnectionPurgingStrategy *strategy*

**Values for strategy**

<table>
<thead>
<tr>
<th>strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lru (default)</td>
<td>Purge least recently used connections.</td>
</tr>
<tr>
<td>lfu</td>
<td>Purge least frequently used connections.</td>
</tr>
<tr>
<td>fifo</td>
<td>Purge oldest connections (first in, first out).</td>
</tr>
<tr>
<td>null</td>
<td>Do not purge connections.</td>
</tr>
</tbody>
</table>

**Description**

Opened connections are added to the connection cache so they can be reused. However, if a process continues to run and these connections are not reused, the cache will continue to grow. Therefore, before each new connection, the cache is checked and purged if it has reached the limit specified by the `-ORBConnectionCacheMax` option or the system default if that option was not used. This option is used to select a cache management strategy that ensures connections are available as needed.

**Usage**

The correct connection cache management strategy depends upon the behavior of the server’s clients. Those that use connections only once will benefit from the FIFO strategy. When pools of connections are used for a while, then not used, the LRU strategy will provide a better means of removing unused connections. The LFU strategy will work best if the connections are used regularly, some more than others. Finally, if the number of connections used by a process can be determined *a priori* and does not exceed the maximum number of simultaneous connections allowed on your system, the null strategy is appropriate to avoid the instantiation of cache management objects.

**Impact**

The use of connection cache management may close connections in use, depending on the algorithm chosen. If the FIFO strategy is selected, for instance, and some old connections are not used for a long time, they may get closed. Subsequent attempts to use these connections will suffer the penalty of reconnection costs.

Purging the cache occurs infrequently, but will occur when a new connection is required and none is available.

**See Also**

20.6.3, 20.6.4

**Example**

dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies: make_TAO_Advanced_Resource_Factory ()
"-ORBConnectionPurgingStrategy fifo"
"-ORBConnectionCachePurgePercentage 50"
20.7.4 ORBInputCDRAssociator lock_type

Values for lock_type

<table>
<thead>
<tr>
<th>lock_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>Access to the allocator for creating and destroying data buffers is synchronized and therefore safe to be used by many threads.</td>
</tr>
<tr>
<td>null</td>
<td>The allocators are not thread safe and may only be used within a single-thread context.</td>
</tr>
</tbody>
</table>

Description

The input CDR allocator is used internally by the ORB to facilitate the decoding and unmarshaling of data received from another ORB. To avoid copying data that is received as arbitrarily large octet sequences, the receiving buffer is passed to the application code that processes it. This transfer of data may or may not involve passing ownership of the data between threads.

Usage

If the data buffer containing the octet sequence is to be shared between threads, access to it and to the allocator responsible for it must be synchronized by using the thread strategy.

If the allocation and the processing of data in the buffer will be achieved within the same thread, greater efficiency and predictability are possible by avoiding the use of mutex locks. This is done by specifying the null strategy.

Note

When using the advanced resource factory, it is possible for this option and the -ORBConnectionCacheLock option to have conflicting values. Both options affect data block locking. If conflicting values are given, the value provided by -ORBConnectionCacheLock will be used. If you use both of these options, they should have the same setting to prevent unexpected results.

See Also

20.2.2, 20.6.2

Example

dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies: _make_TAO_Advanced_Resource_Factory () "-ORBInputCDRAssociator null"
20.7.5 ORBReactorThreadQueue type

Values for type

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lifo (default)</td>
<td>Threads that are waiting in the thread-pool reactor will be selected to run in a last-in-first-out order (LIFO) order.</td>
</tr>
<tr>
<td>fifo</td>
<td>Threads that are waiting in the thread-pool reactor will be selected to run in a first-in-first-out order (FIFO) order.</td>
</tr>
</tbody>
</table>

Description

This option allows an application using the thread-pool reactor to specify the order in which threads should be selected to run by the ACE_Select_Reactor_Token. By default, threads are selected in FIFO order by the ACE_Select_Reactor_Token.

The thread-pool reactor implements the Leader/Followers architectural pattern described in Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2). Briefly, if two or more threads are waiting to run the reactor’s event loop, one thread becomes the leader and gets into the event loop; the remaining threads are followers. When the reactor dispatches an event onto the leader thread, a new leader is selected from among the group of followers, then the leader continues to process the event.

This option controls the order in which a follower thread is selected to become the new leader.

Usage

This option is only usable when the reactor type is the thread-pool (tp) reactor. Either setting should be suitable for any application.

Impact

By using the LIFO reactor thread queue type, multithreaded applications may experience a performance gain by maximizing CPU “cache affinity” by ensuring that the thread waiting the shortest amount of time is selected as the new leader first. However, the LIFO strategy requires an additional data structure in the reactor to hold a list of waiting threads, rather than just using a native operating system synchronization object, such as a semaphore or condition variable.

Results of using one strategy over the other will vary widely across different platforms depending upon the number of CPUs, the amount of CPU cache, and the efficiency of the operating system’s cache management strategy. In addition, if the amount of time each thread spends in an upcall exceeds the time between events, threads will not have to wait for events (i.e., they will usually all be busy processing events), so the selection will have little effect.

See Also

20.7.6
Example
dynamic Advanced Resource Factory Service Object*
TAO_Strategies::make_TAO_Advanced_Resource_Factory () "-ORBReactorThreadQueue fifo"
20.7.6  ORBReactorType reactor_type

Values for reactor_type

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fl</td>
<td>The FL (“fast, light”) toolkit reactor. This reactor is available only when ACE is built with support for the FL toolkit.</td>
</tr>
<tr>
<td>msg_wfmo</td>
<td>The Message Wait For Multiple Objects reactor, available on Win32 systems only.</td>
</tr>
<tr>
<td>select_mt</td>
<td>Multithreaded select reactor. This reactor uses the select() system call to monitor sockets for input and output availability, and employs locks to enforce synchronization between threads.</td>
</tr>
<tr>
<td>select_st</td>
<td>The single-threaded select reactor. This reactor is also based on the select() system call, but assumes it is running in a single thread and therefore does not perform any locking.</td>
</tr>
<tr>
<td>tk_reactor</td>
<td>The Tk toolkit reactor. This reactor is available only when ACE is built with support for the Tk toolkit.</td>
</tr>
<tr>
<td>tp (default)</td>
<td>The thread-pool reactor. This reactor allows for many threads to be used to handle events. If a thread is available to handle an event, the event is dispatched right away. Otherwise it is queued until a thread becomes ready.</td>
</tr>
<tr>
<td>wfmo</td>
<td>The Wait For Multiple Objects reactor, available on Win32 only.</td>
</tr>
</tbody>
</table>

Description

ACE supplies a variety of reactors that may be used to support specialized types of event-driven applications. The reactor is the component of ACE that separates the detection of events from the handling of those events. TAO uses a reactor to accept new connections in a socket context, as well as responding to incoming and outgoing data. The default reactor is perfectly suitable to handle the needs of the ORB in any context.

Usage

Only one reactor type may be specified. If this option is repeated, the last value specified will be used.

Use this option for applications that require a specialized reactor. For example, a Win32 GUI-based application must use the reactor built on the WaitForMultipleObjects() system call to properly integrate with the environment.

Impact

The choice of reactor has very little impact on the application. The only optimization benefit is gained in a single-threaded select-based application. In this configuration, the single-threaded select reactor will remove a single lock from the event-handling loop. Otherwise, the choice of reactor is entirely dependent on the configuration of the application.

See Also

17.3.3.3, 20.6.6
**Example**

dynamic Advanced_Resource_Factory Service Object*

TAO_Strategies::make_TAO_Advanced_Resource_Factory () "-ORBReactorType msg_wfmo"
CHAPTER 21

Server Strategy Factory

21.1 Introduction

Certain elements of the ORB relate only to server-side behavior. In this context, the server is any application that passively accepts connections from other processes and receives requests from those other connections. The server strategy factory is responsible for supporting features of TAO that are specific to servers, including demultiplexing and concurrency-related strategies. The demultiplexing strategies are used to locate POAs and active objects responsible for handling requests. The concurrency strategies control the thread-creation flags and other concurrency-related behaviors.

The server strategy factory is registered with the service configurator using the name Server_Strategy_Factory. TAO’s default server strategy factory is statically registered with the service configurator and is initialized using the static directive. To supply options to the default server strategy factory, add a line, similar to the line shown below, to your service configuration file:

```
static Server_Strategy_Factory "-ORBoption value -ORBoption value ...
```

The service configurator is discussed in greater detail in Chapter 18.
21.2 Interface Definition

Within a TAO application, the server strategy factory is accessed via an interface defined by the base class TAO_Server_Strategy_Factory. Here we show all the public and protected functions of the server strategy factory and describe their use in relationship to the ORB, and the behavior of the default server strategy factory implementation.

The implementation behind this interface is intended to supply parameters to a number of TAO internal objects, such as the ORB core, POA, POA manager, and the various acceptor implementations supporting different pluggable protocols.

Synopsis

```cpp
class TAO_Server_Strategy_Factory : public ACE_Service_Object {
public:
    struct Active_Object_Map_Creation_Parameters {
        Active_Object_Map_Creation_Parameters (void);
        CORBA::ULong active_object_map_size_;
        TAO_Demux_Strategy object_lookup_strategy_for_user_id_policy_;  
        TAO_Demux_Strategy object_lookup_strategy_for_system_id_policy_; 
        TAO_Demux_Strategy reverse_object_lookup_strategy_for_unique_id_policy_;  
        int use_active_hint_in_ids_;  
        int allow_reactivation_of_system_ids_;  

        CORBA::ULong poa_map_size_;  
        TAO_Demux_Strategy poa_lookup_strategy_for_transient_id_policy_;  
        TAO_Demux_Strategy poa_lookup_strategy_for_persistent_id_policy_;  
        int use_active_hint_in_poa_names_;  
    };

    TAO_Server_Strategy_Factory (void)
    virtual ~TAO_Server_Strategy_Factory (void)
    virtual int open (TAO_ORB_Core* orb_core);
    virtual int enable_poa_locking (void);
    virtual int activate_server_connections (void);
    virtual int thread_per_connection_timeout (ACE_Time_Value &timeout);
    virtual int server_connection_thread_flags (void);
    virtual int server_connection_thread_count (void);
    virtual const Active_Object_Map_Creation_Parameters&
    active_object_map_creation_parameters (void) const
protected:
    Active_Object_Map_Creation_Parameters active_object_map_creation_parameters_;
};
```
21.2 Interface Definition

21.2.1 Factory Initialization

The constructor and destructor are defaults and otherwise not interesting. Since the class TAO_Server_Strategy_Factory is derived from ACE_Service_Object, the initialization of the factory is expected to be performed by an implementation of the virtual function ACE_Service_Object::init(int argc, ASYS_CHAR *argv[]). The default server strategy factory implements this function by parsing options provided to it by the service configurator. See below for a complete listing of the options parsed by the default server strategy factory.

To complete the initialization of the server strategy factory, the open() operation is provided:

```c
virtual int open (TAO_ORB_Core* orb_core);
```

The ORB core calls TAO_Server_Strategy_Factory::open() during its initialization, after it has processed its configuration options. This allows all of the service objects that may be dynamically loaded to be initialized before there is any chance that the server strategy factory may need them. The default server factory does nothing in open().

21.2.2 Server Concurrency

An important configuration issue is controlling concurrency-related behavior. The use of threads within an application can increase the capacity of the application, particularly on hosts with multiple processors, but at a cost of increased complexity within the application. The most difficult issue to deal with in a multithreaded application is the synchronization of access to different parts of the application and to data. TAO has been designed with the goal of creating servers that may support hundreds or thousands of clients on very large hosts, as well as creating extremely small, special purpose servers that may have few clients, and must be extremely efficient. To support these different goals, the server strategy factory provides functions to control the concurrency behavior of the server, as well as locks and thread-creation parameters.

```c
virtual int enable_poa_locking (void);
virtual int activate_server_connections (void);
```
These two functions are used by other parts of the ORB to determine how to deal with POA locking and multithreaded servers. If POA locking is to be allowed, then the function `enable_poa_locking()` must return non-zero. Similarly, if the concurrency strategy is to use a thread per connection, then the function `activate_server_connections()` must return non-zero.

The default server factory determines the value of `enable_poa_locking()` based on the `-ORBPOALock` option, returning non-zero if the option is `thread` or zero if `null` is specified. The default server factory returns a non-zero value for `activate_server_connections()` if `-ORBConcurrency` is set to `thread-per-connection`, or zero if `reactive` is specified.

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ORBConcurrency</code></td>
<td>21.3.5</td>
<td>The default value, <code>reactive</code>, specifies that requests will be handled using an event handler model. The thread-per-connection value specifies that a thread will be dedicated to handling requests for each connection established by clients.</td>
</tr>
<tr>
<td><code>-ORBPOALock</code></td>
<td>21.3.7</td>
<td>The default value, <code>thread</code>, specifies that locking is required. The <code>null</code> value specifies that no locking is required.</td>
</tr>
</tbody>
</table>

When the server is configured to use a thread per connection, the service handlers associated with each new connection are made active. That is, a new thread is spawned, which runs code for the handler. This is possible because the connection handlers used in TAO are based on the ACE Active Object pattern, and use the service handler’s `activate()` function to start running in a separate thread. The server strategy factory is responsible for supplying the appropriate parameters to be used in the call to `activate()` through the following functions:

```c
virtual int server_connection_thread_flags (void);
virtual int server_connection_thread_count (void);
```

The default server strategy factory supplies both `THR_BOUND` and `THR_DETACHED` as the values of the server connection thread flags. The
default server factory always returns 1 for the thread count. There is no mechanism provided for using more than one thread per connection.

**Table 21-2 Thread Flags Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBThreadFlags (THR_BOUND</td>
<td>THR_DAEMON</td>
<td>THR_DETACHED</td>
</tr>
</tbody>
</table>

In addition, for the thread-per-connection concurrency model, each thread that is spawned to handle requests arriving on a given connection must be periodically awakened to check for possible ORB shutdown. The server strategy factory provides the following function for this purpose:

```cpp
virtual int thread_per_connection_timeout (ACE_Time_Value &timeout);
```

This function returns -1 if the compile-time default (specified as the preprocessor macro TAO_DEFAULT_THREAD_PER_CONNECTION_TIMEOUT in `$TAO_ROOT/tao/orbconf.h`) should be used; zero if threads should block without checking for ORB shutdown; or a positive non-zero value and sets the timeout parameter with the value specified by the following option:

**Table 21-3 Thread-per-Connection Timeout Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBThreadPerConnectionTimeout {infinite</td>
<td>milliseconds}</td>
<td>21.3.11</td>
</tr>
</tbody>
</table>

**21.2.3 Demultiplexing Strategies**

When resolving a request, the ORB must perform three lookups. The word *demultiplexing* or *demux* is used to describe the act of looking up an object based on an object identifier. The first lookup is to locate the POA identified by the request. The second is a lookup on the POA to find the servant, or active object, that will handle the request. The final lookup is performed within the active object to resolve the function signature requested. The function signature lookup is controlled by the IDL compiler via the -H
command line option (see 5.10), and will generally use the *perfect hashing* algorithm as generated by the GNU `gperf` application.

TAO provides three configurable strategies to servers for demultiplexing POAs and servants: the Linear Search strategy, the Dynamic Hash strategy, and the Active Demux strategy.

The server strategy factory is not directly responsible for providing the demultiplexing operations; rather it supplies the configuration information to the parts of the ORB that maintain the tables used to demultiplex components of requests. These parameters are supplied by the following function:

```cpp
virtual const Active_Object_Map_Creation_Parameters&
active_object_map_creation_parameters (void) const
```

The value returned from this function is a structure that encapsulates all of the state information needed to construct the desired demultiplexing strategies. The default strategy factory populates the active object map creation parameters by interpreting two sets of options, the POA Map options and the Active Object Map options. The strategy used within an active object to resolve a particular function signature is determined by the IDL compiler, and therefore is not included as part of the active object map creation parameters structure.

### 21.2.3.1 Linear Search

The linear search strategy is the simplest demultiplexing strategy. Entities are stored in an unsorted array. As elements are removed from the list, the position of the hole is retained in a free list. New elements added to the list will fill these holes, or be appended to the list. Locating elements of the list requires examining the key value of each element in the list sequentially, until the matching element is found.

The cost of adding a new element to the linear search map is small, as no calculations are performed to convert the key value from one form to another.

Another point to consider is the cost of growing the list. As described below, it is possible to specify an initial size for the active object maps and for the POA maps. If the number of elements in the map extends beyond this size, the map will grow. This growth involves allocating twice as much space as the previous map contained, duplicating each existing element, using its copy...
constructor, finally releasing the old space by calling each element’s
destructor.

When comparing relative performance of the linear search strategy versus
other strategies, the lookup time is $O(n/2)$ on average, and $O(n)$ worst case.
The linear search strategy does not scale well to large numbers of objects.

### 21.2.3.2 Dynamic Hash

The dynamic hash strategy is substantially more efficient than the linear
search for handling larger numbers of elements. This strategy uses a hash table
to achieve a nearly constant lookup time regardless of the size of the table.
Occasionally two or more keys will result in the same hash table index. Using
a sufficiently large table will mitigate this, but when such “collisions” occur,
the elements sharing the table index are searched linearly.

Compared to the linear search strategy, there is a greater constant cost for each
access of the map. The key value must be hashed before it may be used to
insert or locate an element. The hashing algorithm used for object identifiers
and POA names is encoded in `ACE::hash_pjw()`.

Unlike the linear search strategy, or the active demux strategy, the map size of
the dynamic hash remains fixed throughout the life of the process. If the
selected map size is smaller than the actual number of elements stored,
hashing collisions will occur, resulting in degraded lookup performance.

### 21.2.3.3 Active Demux

For the greatest performance in demultiplexing, TAO uses the active
demultiplexing strategy. Active demultiplexing takes advantage of the ACE
Active Map, that uses array indices and object “generations” as the search key.
The effect of this is that searching the active map involves only a single array
dereference. Active maps are optimized for storage, reusing slots made
available by the removal of earlier entries. To guard against incorrect
references, entries in an active map have a generation, which is the count of
the number of times a particular slot has had a value assigned to it. For this
reason, active demultiplexing is available only when an object reference is
transient and guaranteed not to be reused. The size of the active map will
increase over time in the same manner as with the linear search strategy.

In case of persistent objects, or objects with references that may be reused,
TAO provides an extension to the standard object reference demultiplexing.
An active “hint” may be added to the POA name or the object key that is used to attempt to locate an object in a secondary active map. It is not guaranteed that the hint is valid, and if it is not, then the dynamic hash table or linear search will resolve the reference. The use of active hints in object identifiers or POA names will increase the size of each map accordingly.

### 21.2.4 POA Map Options

All POAs within a server must register themselves with the root POA. The root POA submits these registrations to the object adapter, which adds them to the appropriate map. The TAO object adapter maintains two separate maps, one for POAs with persistent IDs and one for POAs with transient IDs. To construct its maps, the object adapter obtains a reference to the `struct Active_Object_Map_Creation_Parameters` and refers to the following values:

```c
CORBA::ULong poa_map_size;
TAO_Demux_Strategy poa_lookup_strategy_for_transient_id_policy;
TAO_Demux_Strategy poa_lookup_strategy_for_persistent_id_policy;
int use_active_hint_in_poa_names;
```

The default server strategy factory options that set these values are shown in this table.

**Table 21-4 POA Map Creation Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`-ORBActiveHintInPOANames [1</td>
<td>0]`</td>
<td>21.3.2</td>
</tr>
<tr>
<td>`-ORBPersistentIDPolicyDemuxStrategy [dynamic</td>
<td>linear]`</td>
<td>21.3.6</td>
</tr>
<tr>
<td><code>-ORBPOAMapSize size</code></td>
<td>21.3.8</td>
<td>Sets the initial number of entries in the POA lookup tables. This sets the value of the parameter <code>poa_map_size</code>. Default is 24.</td>
</tr>
</tbody>
</table>
21.2 Interface Definition

21.2.5 Active Object Map Parameters

Once a POA is resolved, it must locate an active object that will be used to handle the request. Each POA contains maps of active objects, one for objects created using the system ID policy, and another one for objects created using the user ID policy. When the unique ID policy is used, a map is created for performing reverse lookups of ID, based on the object adapter.

To construct its maps, the POA obtains a reference to the `struct Active_Object_Map_Creation_Parameters` and refers to the following values:

- `CORBA::ULong active_object_map_size_;
- TAO_Demux_Strategy object_lookup_strategy_for_user_id_policy_;
- TAO_Demux_Strategy object_lookup_strategy_for_system_id_policy_;
- TAO_Demux_Strategy reverse_object_lookup_strategy_for_unique_id_policy_;
- int use_active_hint_in_ids_;  
- int allow_reactivation_of_system_ids_;

The default server strategy factory options that set these values are shown in Table 21-5.

### Table 21-5 Active Object Map Creation Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBActiveHintInIDs {1</td>
<td>0}</td>
<td>21.3.1</td>
</tr>
<tr>
<td>-ORBActiveObjectMapSize size</td>
<td>21.3.3</td>
<td>Sets the initial number of entries in the active object lookup tables. Default is 64. Sets the value of <code>active_object_map_size_</code>.</td>
</tr>
</tbody>
</table>
The remainder of this chapter describes the individual options interpreted by the default server strategy factory supplied with TAO. These options are supplied to the default server strategy factory by the service configurator, through the use of the `static` initialization directive (see 18.3).

### 21.3 Default Server Strategy Factory Options

The remainder of this chapter describes the individual options interpreted by the default server strategy factory supplied with TAO. These options are supplied to the default server strategy factory by the service configurator, through the use of the `static` initialization directive (see 18.3).

#### Table 21-5 Active Object Map Creation Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBAAllowReactivationOfSystemIDs {1</td>
<td>0}</td>
<td>21.3.4</td>
</tr>
<tr>
<td>-ORBSystemIDPolicyDemuxStrategy {active</td>
<td>dynamic</td>
<td>linear}</td>
</tr>
<tr>
<td>-ORBUniqueIDPolicyReverseDemuxStrategy {dynamic</td>
<td>linear}</td>
<td>21.3.13</td>
</tr>
<tr>
<td>-ORBUserIDPolicyDemuxStrategy {dynamic</td>
<td>linear}</td>
<td>21.3.14</td>
</tr>
</tbody>
</table>
21.3.1 ORBActiveHintInIDs \textit{enabled}

\textbf{Values for enabled}

\begin{tabular}{ll}
1 (default) & Hints are to be used in object IDs. \\
0 & Hints are not to be used in object IDs. \\
\end{tabular}

\textbf{Description}  This option controls the embedding of additional information in the object reference for use as a hint in resolving object references. These hints may allow for faster resolution of object references. When the POA policy USER_ID is used in creating new object IDs, TAO is not able to use its active demux strategy based solely on the ID. This can have an impact on performance, as the number of active objects bound to a POA increases. By allowing TAO to concatenate some additional information to the object ID, a secondary active map may be used to look up the object reference. As long as the hint is valid, the associated object will be located, using the fast lookup of the active demux strategy. If the hint is no longer valid, perhaps from the active object having been destroyed, then reconstructed, the lookup strategy otherwise defined will be used to locate the object.

\textbf{Usage}  Add active hint information to the object ID and use this information to more efficiently look up the servants in the POA.

\textbf{Impact}  The use of active hints as an additional component of the object ID portion of the IOR will generally reduce object lookup time, in some cases substantially so. The cost of doing so is in the form of space. The hint requires an extra eight bytes in the IOR. An active map is used to facilitate the lookup, based on the hint. This is in addition to the secondary lookup method, either a dynamic hash or linear search table.

Active lookup tables are reallocated when space is required to store more entries than is available.

When used in conjunction with `-ORBAAllowReactivationOfSystemIDs` with a value of 1, it provides predictable latency regardless of the number of object IDs in a POA.

\textbf{See Also}  21.3.3

\textbf{Example}  \texttt{static Server\_Strategy\_Factory \"-ORBActiveHintInIDs 0\"}
21.3.2 ORBActiveHintInPOANames *enabled*

**Values for enabled**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (default)</td>
<td>Hints are to be used in POA names.</td>
</tr>
<tr>
<td>0</td>
<td>Hints are not to be used in POA names.</td>
</tr>
</tbody>
</table>

**Description**
Adding active hints to the POA names embedded in the object key may allow for faster lookup of POAs by the ORB core. The use of hints in POA names is turned on by default.

**Usage**
Turn ON this option when predictability/performance is important, and you are using multiple POAs.

Turn OFF this option to minimize memory usage.

**Impact**
The active hints passed in the object key facilitate quick look up of POA names, and make the lookup time predictable for any level of nested POAs.

Active hints result in larger IORs and extra server-side state. They also increase the request size (typically by eight bytes) since additional hint information is added to the POA name in the object key.

When used in conjunction with active demuxing strategy for POA lookups, active hints provide predictable latency regardless of the number and nesting of POAs.

**See Also**
21.3.8

**Example**
```
static Server_Strategy.Factory "-ORBActiveHintInPOANames 0"
```
21.3 Default Server Strategy Factory Options

21.3.3 ORBActiveObjectMapSize map_size

Description  
This option specifies the initial number of entries in the active object map. By default, the active object map is initialized with sixty-four (64) entries if no map size is specified. The default active object map size is specified as the preprocessor macro TAO_DEFAULT_SERVER_ACTIVE_OBJECT_MAP_SIZE in $TAO_ROOT/tao/orbconf.h. This value affects all of the active object maps and hint tables. When the dynamic hash strategy is used for SYSTEM_ID or USER_ID active objects, the value specified here determines the total number of hash table buckets. When using either the linear search or active demux strategies, this value specifies the initial number of entries in the map.

Usage  
If any POA in a server is going to manage more than the default number of active objects, then a larger number should be specified. On the other hand, if memory conservation is important, and the maximum number of active objects on any POA is going to be less than the default, then specify a smaller number.

Impact  
Setting the initial map size too small degrades the predictability of the application. This degradation takes the form of map resizing when linear or active demux strategies are used, and inefficient lookups when the dynamic strategy is used.

This option controls the initial size of the maps for user-specified object IDs, system-generated object IDs, and the reverse lookup map. Setting this value larger than the number of active objects in the system wastes memory for linear and active demux strategies. When using the dynamic demux strategy, increasing the size of the map may improve performance.

See Also  
21.3.1, 21.3.9

Example  
static Server_Strategy_Factory "-ORBActiveObjectMapSize 100"
21.3.4 ORB.Allow.ReactivationOfSystemIDs enabled

Values for enabled

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (default)</td>
<td>Enables the reuse of previous system-generated object IDs.</td>
</tr>
<tr>
<td>0</td>
<td>Guarantees the uniqueness of system-generated object IDs.</td>
</tr>
</tbody>
</table>

Description
This option controls whether system-generated object IDs can be reactivated. When enabled, it allows an ID that was generated by the system to be reused after it has been deactivated. This would typically occur when clients hold references to objects whose servants are deactivated. Further use of the reference would force the server to activate a servant with the old ID.

Usage
Servers that use system-generated IDs in an environment where the lifespan of a server (or servant) may be less than the lifespan of its clients should enable this option.

Override the default value to disable the reuse of system ids when it is known that no client will want to reactivate a previously-deactivated servant with a system-generated id.

Impact
When disabled, the IORs can be shortened, an extra comparison in the critical upcall path removed, and some memory on the server side can be saved. This allows the use of active demultiplexing as the primary lookup strategy for resolving object references.

The reuse of object IDs presents a problem in that it limits TAO’s ability to use the active demultiplexing strategy with system-generated IDs. This situation requires the use of hints to obtain the greatest performance.

See Also
21.3.1, 21.3.9

Example
static Server_Strategy_Factory "-ORB.Allow.ReactivationOfSystemIDs 0"
21.3 Default Server Strategy Factory Options

## 21.3.5 ORBConcurrency strategy

### Values for strategy

<table>
<thead>
<tr>
<th>strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive (default)</td>
<td>The ORB’s reactor is used to reactively process incoming requests from all connections.</td>
</tr>
<tr>
<td>thread-per-connection</td>
<td>A new thread is spawned to service each connection.</td>
</tr>
</tbody>
</table>

### Description

This option specifies the concurrency strategy used by the ORB to control server behavior. When multiple clients attempt to simultaneously connect to a server, the response time for each client may suffer. The problem is made even worse when each request takes a long time to execute. Response time may be improved by using multiple threads in the server. However, when dealing with legacy code that is not thread safe, multithreading may not be possible.

### Usage

The default value, reactive, is appropriate when requests take a fixed, relatively uniform amount of time and are largely I/O bound. The reactor type used by the ORB is controlled by the `-ORBReactorType` option. Each thread that calls `ORB::run()` or `ORB::perform_work()` can be used by the reactor to process requests. The thread pool reactor must be specified to effectively use multiple threads in the reactive mode.

The use of `thread-per-connection` is appropriate when there are multiple connections active simultaneously and the clients are set up to use only one request per connection. It should not be used in situations where the time required to service a request is small compared to the time required to create a thread. The use of `thread-per-connection` requires that data shared by servants be locked and common functions be thread safe.

### Impact

When using `thread-per-connection`, a new thread is created for each connection made. This allows for more prompt servicing of new connections generally, but adds the cost of thread creation for every new connection.

### See Also

17.3.3.2, 21.3.11, 20.7.6

### Example

```
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection"
```
### 21.3.6 ORBPersistentIDPolicyDemuxStrategy strategy

#### Values for strategy

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic (default)</td>
<td>Use a dynamic hashing strategy.</td>
</tr>
<tr>
<td>linear</td>
<td>Use a linear search strategy.</td>
</tr>
</tbody>
</table>

#### Description

This option selects the algorithm the ORB core uses for locating a persistent POA, based on an object request. The active demultiplexing strategy is not available for POAs identified with the PERSISTENT policy. Persistent POAs are intended to outlive the server process, and active demultiplex index values are valid only during the life of the process that generates them. An active hint may be used along with the object id to benefit from the performance boost of active demultiplexing.

#### Usage

If the number of persistent POAs is small, and a design goal is to minimize the memory footprint of a server, the linear search strategy will provide reasonable performance with the smallest memory footprint.

#### Impact

The dynamic hashing strategy incurs the additional lookup cost of hashing the POA name.

The linear search strategy quickly degrades lookup performance as the number of persistent POAs increases and results in additional performance degradation as the addition of new persistent POAs forces the map to resize.

#### See Also

21.3.2, 21.3.8, 21.3.12

#### Example

```
static Server_Strategy_Factory "-ORBPersistentIDPolicyDemuxStrategy linear"
```
21.3 Default Server Strategy Factory Options

21.3.7 ORBPOALock lock_type

Values for lock_type

<table>
<thead>
<tr>
<th>lock_type</th>
<th>Description</th>
<th>Usage</th>
<th>Impact</th>
<th>See Also</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread (default)</td>
<td>Specifies inter-thread mutex to guarantee exclusive access.</td>
<td>The value null removes the processing burden of acquiring a lock when the POA is not shared by separate threads.</td>
<td>Removing the locking overhead when the POA is going to be accessed only by a single thread improves performance.</td>
<td>21.3.5</td>
</tr>
<tr>
<td>null</td>
<td>No locking.</td>
<td>The value thread is appropriate for all other situations.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

static Server_Strategy_Factory "-ORBPOALock null"
**21.3.8 ORBPOAMapSize map_size**

**Description**  
This option specifies the initial number of entries in the POA map. The default value is twenty-four (24), as specified by the preprocessor macro TAO_DEFAULT_SERVER_POA_MAP_SIZE in $TAO_ROOT/tao/orbconf.h. This value affects all of the POA maps and hint tables. When the dynamic hash strategy is used for PERSISTENT or TRANSIENT POAs, the value specified here determines the total number of hash table buckets. When using either the linear search or active demux strategies, this value specifies the initial number of entries in the map.

**Usage**  
If any ORB in a server is going to have more than the default number of POAs, then a larger number should be specified. On the other hand, if memory conservation is important, and the maximum number of POAs is going to be less than the default number, it is possible to use a smaller number.

**Impact**  
Setting the initial map size too small degrades the predictability of the application. This degradation takes the form of map resizing when linear or active demux strategies are used, and inefficient lookups when the dynamic strategy is used.

This option controls the initial size of the maps for both transient and persistent POAs. Setting this value larger than the number of POAs in the system wastes memory for linear and active demux strategies. When using the dynamic demux strategy, increasing the size of the map may improve performance.

**See Also**  
21.3.2, 21.3.12

**Example**  
static Server_Strategy_Factory "-ORBPOAMapSize 10"
21.3.9 ORBSystemIDPolicyDemuxStrategy strategy

Values for strategy

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>Use the active demultiplexing strategy.</td>
</tr>
<tr>
<td>dynamic</td>
<td>Use the dynamic hashing strategy.</td>
</tr>
<tr>
<td>linear</td>
<td>Use the linear search strategy.</td>
</tr>
</tbody>
</table>

Description

This option defines the demultiplexing strategy used by the POA for managing active objects with system-generated object identifiers. The active demultiplexing strategy provides the smallest lookup time per request, but at the cost of an eight-byte identifier. Dynamic hashing provides performance that is nearly as good as the active demux, but using a four-byte identifier. The linear search strategy provides reasonable performance for small maps, using a four-byte identifier, but does not scale very well as the number of active objects increases.

Usage

Selecting the appropriate strategy involves determining the number objects a POA must service and the cost associated with an eight-byte identifier versus a four-byte identifier.

Impact

The cost associated with the active demultiplexing strategy is a larger object key and the occasional resizing of the map if the initial size is too small. The cost of the dynamic hashing strategy is degraded performance as the number of objects increases. The cost of the linear search is quickly-degraded performance as the number of objects increases and the possible resizing of the map as additional active objects are added.

See Also

21.3.3

Example

static Server_Strategy_Factory "-ORBSystemIDPolicyDemuxStrategy dynamic"
21.3.10 ORBThreadFlags *flags*

**Values for flags**

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THR_BOUND</td>
<td>Each new thread is bound to a LWP, allowing for each to be</td>
</tr>
<tr>
<td></td>
<td>scheduled separately.</td>
</tr>
<tr>
<td>THR_DAEMON</td>
<td><em>Not available for Win32 based platforms.</em> The <code>wait()</code></td>
</tr>
<tr>
<td></td>
<td>function cannot be invoked on daemon threads.</td>
</tr>
<tr>
<td>THR_DETACHED</td>
<td>Threads do not return status when they terminate.</td>
</tr>
<tr>
<td>THR_NEW_LWP</td>
<td>Creates a new light weight process for thread processing.</td>
</tr>
<tr>
<td>THR_SUSPENDED</td>
<td>Creates new threads in the “suspended” state.</td>
</tr>
</tbody>
</table>

**Description**

This option specifies the flags to use when creating new threads in a server. These thread creation flags are used when the concurrency strategy is specified as `thread-per-connection`. In that case, the service handlers activated when a new connection is established are supplied these flags by the connection acceptor. By default, new threads are created with both the THR_BOUND and THR_DETACHED flags.

**Usage**

Multiple thread flags may be specified as a single value by combining the desired flags using a vertical bar (`|`) between the flags. No spaces may appear between the values and the `|`. For example:

```
-ORBThreadFlags THR_BOUND|THR_NEW_LWP
```

**Impact**

This option only has an effect when the concurrency strategy is set to `thread-per-connection`. The specific impact to your system depends on your operating system’s threading capability.

If there is a limit to the number of lightweight processes that may be created, the use of THR_NEW_LWP can fail if your process exceeds that value.

An application may require that threads be invoked by a scheduler. To achieve this goal in an environment where requests are received asynchronously, the THR_SUSPENDED flag allows newly-created threads to wait to be started by a scheduler. This, of course, requires an application configured to resume all suspended threads appropriately.

**See Also** 21.3.5

**Example**

```
static Server_Strategy_Factory "-ORBThreadFlags THR_BOUND|THR_SUSPENDED"
```
21.3 Default Server Strategy Factory Options

21.3.11 ORBThreadPerConnectionTimeout time

Values for time

<table>
<thead>
<tr>
<th>value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinite</td>
<td>Threads spawned to handle requests for each client connection will never check for ORB shutdown.</td>
</tr>
<tr>
<td>n &gt; 0</td>
<td>Threads spawned to handle requests for each client connection will check for ORB shutdown every n milliseconds (default is 5000).</td>
</tr>
</tbody>
</table>

Description When using the thread-per-connection concurrency strategy for a server, the ORB will spawn a new thread for each new connection, and that thread will be dedicated to handling incoming requests over its connection. These threads will remain active as long as the connection remains established. However, since these threads do not normally participate in the ORB’s reactor, they need a way to determine if the ORB has been shut down. A timer can be used to wake up each thread on a specific time interval to check to see if the ORB has shutdown. The default time interval is indicated by the preprocessor macro TAO_THREAD_PER_CONNECTION_TIMEOUT, defined in $TAO_ROOT/tao/orbconf.h as 5000 milliseconds (5 seconds). You can use this server strategy factory option to either disable the timeout or change its value.

Usage Timeout after 1 second.

-ORBThreadPerConnectionTimeout 1000

Do not timeout.

-ORBThreadPerConnectionTimeout infinite

Impact This option only has an effect when the concurrency strategy is set to thread-per-connection. The specific impact to your system depends on your operating system’s threading capability.

See Also 21.3.5

Example static Server_Strategy_Factory "-ORBThreadPerConnectionTimeout infinite"
21.3.12 ORBTransientIDPolicyDemuxStrategy strategy

Values for strategy

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>Use the active demultiplexing strategy.</td>
</tr>
<tr>
<td>dynamic</td>
<td>Use the dynamic hashing strategy.</td>
</tr>
<tr>
<td>linear</td>
<td>Use the linear search strategy.</td>
</tr>
</tbody>
</table>

Description

This option defines the demultiplexing strategy used by the ORB core for managing transient POAs. The active demultiplexing strategy provides the smallest lookup time per request, but at the cost of an eight-byte identifier. Dynamic hashing provides performance that is nearly as good as the active demux, but using a four-byte identifier. The linear search strategy provides reasonable performance for small maps, using a four-byte identifier, but does not scale very well as the number of active objects increases.

Usage

Selecting the appropriate strategy involves determining the number of transient POAs that will be mapped and the cost associated with an eight-byte identifier versus a four-byte identifier.

Impact

The active demultiplex strategy results in a larger object key and the occasional resizing of the map if the initial size is too small. The hashing strategy degrades performance as the number of objects increases. The linear search strategy quickly degrades lookup performance as the number of objects increases and results in additional performance degradation when the addition of new active objects forces the map to resize.

See Also

21.3.2, 21.3.8

Example

static Server_Strategy_Factory "-ORBTransientIDPolicyDemuxStrategy dynamic"
21.3.13 ORBUniqueIDPolicyReverseDemuxStrategy **strategy**

**Values for strategy**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic (default)</td>
<td>Use a dynamic hash algorithm.</td>
</tr>
<tr>
<td>linear</td>
<td>Use a linear search algorithm.</td>
</tr>
</tbody>
</table>

**Description**

This option defines the strategy used by a POA to locate an object identifier, based on a servant reference. This reverse lookup is possible only when the POA’s uniqueness policy is set to `UNIQUE_ID`, meaning that each object ID will be associated with a different servant. If `MULITPLE_ID` is selected as the uniqueness policy, there can be many object identifiers associated with a servant, so a reverse lookup is not possible.

**Usage**

The default strategy is generally better than the linear search strategy. This is because the hashing algorithm for servants is simply to convert a pointer to the servant to an unsigned long. In a case of a highly-optimized environment, where the dynamic mapping classes are compiled out of ACE, then the linear search must be used.

**Impact**

Space is allocated for a servant-to-object-ID map only when the POA’s uniqueness policy is set to `UNIQUE_ID`. The amount of space initially allocated for the reverse lookup table is the same as the active object map size.

**See Also**

21.3.3

**Example**

```
static Server_Strategy_Factory "-ORBUniqueIDPolicyReverseDemuxStrategy linear"
```
21.3.14 ORBUserIDPolicyDemuxStrategy strategy

Values for strategy

<table>
<thead>
<tr>
<th>strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>Use a dynamic hash algorithm.</td>
</tr>
<tr>
<td>linear</td>
<td>Use a linear search algorithm.</td>
</tr>
</tbody>
</table>

Description

This option defines the demultiplexing strategy used by the POA for managing active objects with user-specified object identifiers. Dynamic hashing provides very good lookup performance at the expense of hashing the object identifier. The linear search strategy provides reasonable performance for small maps, but does not scale very well as the number of active objects increases. Active demultiplexing is not an option with user-specified object identifiers, but an active hint may be used along with the native strategy for increased performance.

Usage

Selecting the appropriate strategy involves determining the number of objects a POA must service and the cost associated with hashing the object identifiers versus iterating through a list comparing object identifiers.

Impact

The dynamic hashing strategy incurs the additional lookup cost of hashing the object identifier.

The linear search strategy quickly degrades lookup performance as the number of objects increases, and results in additional performance degradation as the addition of new active objects forces the map to resize.

See Also

21.3.1, 21.3.3

Example

static Server_Strategy_Factory "-ORBUserIDPolicyDemuxStrategy linear"
22.1 Introduction

The client strategy factory supports those elements of TAO that are specific to the behavior of clients. A client, for this discussion, is any application that actively establishes connections to other processes, sends requests, and perhaps receives replies. The client strategy factory provides control over several resources used by clients. For example, the factory supplies the mechanism employed when waiting for a response from a server. The client strategy factory also supplies a transport multiplexor that enables multiple asynchronous requests across a single connection. All of these objects are used by TAO internally and are not intended to be used by application code.

The client strategy factory is registered with the service configurator using the name `Client_Strategy_Factory`. TAO’s default client strategy factory is statically registered with the service configurator and is initialized using the `static` directive. To supply options to the default client strategy factory, add a line, similar to the line shown below, to your service configuration file:

```
static Client_Strategy_Factory "-ORBoption value -ORBoption value ...
```
The service configurator is discussed in greater detail in Chapter 18.

22.2 Interface Definition

Within a TAO application, the client strategy factory is accessed via an interface defined by the class TAO_Client_Strategy_Factory. Here we show all of the public functions of the client strategy factory, describe their use in relationship to the ORB, and the behavior of the default client strategy factory implementation.

Synopsis

class TAO_Export TAO_Client_Strategy_Factory : public ACE_Service_Object
{
public:
   TAO_Client_Strategy_Factory (void);
   virtual ~TAO_Client_Strategy_Factory (void);

   virtual ACE_Lock* create_profile_lock (void);
   virtual TAO_Transport_Mux_Strategy* create_transport_mux_strategy (TAO_Transport* transport);
   virtual TAO_Wait_Strategy* create_wait_strategy (TAO_Transport* transport);
   virtual int allow_callback (void);
   virtual TAO_Connect_Strategy *create_connect_strategy (TAO_ORB_Core *);
   virtual ACE_Lock* create_ft_service_retention_id_lock (void);
};

The create_ft_service_retention_id_lock() function is used to return a lock to TAO’s fault tolerance service. Its behavior is not affected by any options provided by the client strategy factory.

22.2.1 Profile Locking

CORBA defines a generic format called the Interoperable Object Reference (IOR) to identify objects. An object reference identifies a CORBA object and one or more paths through which the object can be accessed. Each path references one server location that implements the object, and an opaque identifier valid on that particular server. References to server locations are called profiles. An object may have multiple server locations to support load balancing, fault tolerance, or other quality-of-service-specific optimizations.

Usually, an IOR’s state is accessed in a read-only fashion by the client-side ORB. However, if the client receives a LOCATION_FORWARD reply message from a server, the client-side ORB can modify the IOR’s state to update the affected profiles. Without appropriate synchronization, multiple threads can
modify the IOR’s internal profile information, potentially leading to corruption of the IOR’s internal state or to inconsistent views of that state by other threads.

The profile lock interface is used internally by the stub to protect access to profile objects. A profile is an abstraction of the endpoint of a connection. For instance, the profile of an IIOP connection is defined by an open socket. Occasionally a profile in use may have to change, particularly if the location of a remote object changes. A lock is needed to protect against multiple threads attempting to modify the stub’s profile. The client strategy factory supplies this lock through the following function:

```cpp
virtual ACE_Lock* create_profile_lock (void);
```

The default client strategy factory creates a lock of the type specified by the ORBProfileLock option, shown in Table 22-1.

**Table 22-1 Profile Locking Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBProfileLock {thread</td>
<td>null}</td>
<td>22.3.2 Determines the type of profile lock created by the default client strategy factory.</td>
</tr>
</tbody>
</table>

The default behavior is to create a thread mutex. This enables a stub to be used by many threads in a client. If a client is single threaded, or all processing related to a single stub is performed in a single thread, use of a null mutex is acceptable and may provide better performance.

### 22.2.2 Transport Multiplexing Strategies

Transport multiplexing can optimize the use of a single connection between a client and a server for many concurrent requests. There are currently two kinds of transport multiplexing strategies defined: exclusive and multiplexed.

An exclusive transport uses a connection to service a single request and receive a reply before making it available for another request. A multithreaded client using the exclusive transport strategy will open a new connection for each concurrent request. The multiplexed strategy allows multiple concurrent requests to share a connection, using asynchronous callbacks to handle the distribution of the replies.
The transport multiplexing strategy is used internally by the transport portion of the communications protocol. The transport portion handles open connections and is defined as part of the pluggable protocols framework:

```cpp
virtual TAO_Transport_Mux_Strategy *create_transport_mux_strategy (TAO_Transport *transport);
```

The default client strategy factory supplies the appropriate type of strategy object, based on the option supplied to it by the service configurator. The option for selecting the multiplexing strategy is listed in Table 22-2.

**Table 22-2 Transport Multiplexing Strategy Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBTransportMuxStrategy</td>
<td>22.3.3</td>
<td>Determines the multiplexing strategy object created by the default client strategy factory.</td>
</tr>
</tbody>
</table>

### 22.2.3 Wait Strategies

Wait strategy objects are used by the transport layer in the client to control the client’s behavior while the client is anticipating a response from a server, after invoking a synchronous request. The wait strategy is also known as the client connection handler. The client strategy factory interface is required to return a wait strategy object for use by the protocol transport object:

```cpp
virtual TAO_Wait_Strategy *create_wait_strategy (TAO_Transport *transport);
```

The default client strategy factory will return a wait strategy object that is one of the following four specializations: *wait-on-read*, *wait-on-reactor*, *wait-on-leader-follower*, or *wait-on-leader-follower-no-upcall*.

Wait-on-read is the simplest implementation. There is no locking and no reactor is used for the reply. When a request is sent to the server, this wait strategy simply blocks until it has read the reply. This strategy does not support handling new requests, and hence nested upcalls, while waiting for the reply from the server. This strategy provides the lowest possible overhead for a multithreaded environment. In a single-threaded client this strategy has an adverse affect on performance, as the process would block on every call. Wait-on-read corresponds to the `rw` option.

Wait-on-reactor is a very effective strategy for single-threaded clients. With this model, the transport registers with the reactor and uses an event handler...
internally to wait for a response. This strategy should only be used for single-threaded clients, as the threads must share access to the reactor. Wait-on-reactor corresponds to the \texttt{st} option.

Wait-on-leader-follower is the strategy intended for use in reactor-based multithreaded clients. This strategy enables multiple threads to register with a reactor and use an event handler for processing the reply when it is available. Wait-on-leader-follower corresponds to the \texttt{mt} option, and is the default.

The wait-on-leader-follower-no-upcall strategy combines the features of the wait-on-read and wait-on-leader-follower strategies. Like the wait-on-leader-follower strategy, it allows multiple threads to register with a reactor and participate in the \textit{leader-follower} interaction described in 17.4.4.3. However, like the wait-on-read strategy, it does not allow these threads to handle nested upcalls. Wait-on-leader-follower-no-upcall corresponds to the \texttt{mt_noupcall} option.

\textbf{Note} \textit{The wait-on-leader-follower-no-upcall wait strategy is an experimental wait strategy. It was designed specifically for and has been thoroughly tested in the use cases presented in 17.4.4.4. It has not been exercised in a wide variety of other use cases. Please use caution and due diligence in testing your application's behavior with this option if you decide to use it.}

More information on the effect of wait strategy on threading behavior is provided in 17.3.3.2 and 17.4.4.

The client strategy factory interface defines another function that indicates whether callbacks should be allowed:

\begin{verbatim}
virtual int allow_callback (void);
\end{verbatim}

This function returns zero when the wait-on-read wait strategy is used. It returns a non-zero value otherwise. The wait-on-read strategy will still allow callbacks as long as the application is multithreaded. This function is currently only used to enable special optimizations when using SHMIOP in combination with the wait-on-read wait strategy.

The default client strategy factory supplies the appropriate type of strategy object, based on the option supplied to it by the service configurator. By using the \texttt{ORBWaitStrategy} option, shown in Table 22-3, the choice of a wait strategy is deferred until run time.
### Connect Strategies

Connect strategy objects are used by the transport layer in the client to control the client’s behavior while initiating a connection to a server. The client strategy factory interface is required to return a connect strategy object for use by the transport-protocol-specific connector object:

```c
virtual TAO_Connect_Strategy *create_connect_strategy (TAO_ORB_Core *orb_core);
```

The default client strategy factory will return a connect strategy object that is one of the following three specializations: blocking, reactive, or leader-follower.

The blocking connect strategy is the simplest implementation. There is no locking and no reactor is used to wait for the connection to succeed or fail. When a connection is attempted to a server, this connect strategy simply blocks until the connection attempt completes successfully or fails with an error. This strategy provides the lowest possible overhead for a multithreaded environment. In a single-threaded client this strategy may have an adverse affect on performance, as the process would block each time a new connection is established. The blocking connect strategy corresponds to the `blocked` option.

---

**Table 22-3 Wait Strategy Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ORBWaitStrategy</code> {mt</td>
<td>st</td>
<td>rw</td>
</tr>
</tbody>
</table>

---

**Note**

The `-ORBWaitStrategy` option is the same as the `-ORBClientConnectionHandler` option. The use of the `-ORBClientConnectionHandler` option has been deprecated.

---

**Note**

If you use the wait-on-read `(rw)` wait strategy option, you should also use the exclusive transport multiplexing option. Otherwise, unexpected results may occur. See 22.2.2.
The reactive connect strategy is a very effective strategy for single-threaded clients. With this model, the connector registers with the reactor and uses an event handler internally to wait for the connection attempt to complete. The thread is not blocked while the connection is being attempted. This strategy should only be used for single-threaded clients, as the threads must share access to the reactor. The reactive connection strategy corresponds to the reactive option.

The leader-follower connect strategy is intended for use in reactor-based multithreaded clients. This strategy enables multiple threads to register with a reactor and use an event handler for dealing with completed connection attempts. The leader-follower connect strategy corresponds to the lf option, and is the default.

More information on the effect of connect strategy on threading behavior is provided in 17.4.1.

The default client strategy factory supplies the appropriate type of strategy object, based on the option supplied to it by the service configurator. By using the ORBConnectStrategy option, shown in Table 22-4, the choice of a connect strategy is deferred until run time.

Table 22-4 Connect Strategy Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBConnectStrategy {lf</td>
<td></td>
<td>Determines the type of connect strategy object created by the default client</td>
</tr>
<tr>
<td>reactive</td>
<td>blocked}</td>
<td>strategy factory.</td>
</tr>
</tbody>
</table>

22.3 Client Strategy Factory Options

The remainder of this chapter describes the individual options interpreted by the default client strategy factory. These options are supplied to the default client strategy factory by the service configurator through the use of a static initialization directive (see 18.3).
22.3.1 ORBConnectStrategy connect_type

Values for handler_type

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1f (default)</td>
<td>Use the multithreaded connect strategy, which is based on the leader-follower model.</td>
</tr>
<tr>
<td>reactive</td>
<td>Use the reactive connect strategy, which allows non-blocking connects in a single-threaded client.</td>
</tr>
<tr>
<td>blocked</td>
<td>Use the blocking connect strategy. Client threads block while initiating a connection with a server.</td>
</tr>
</tbody>
</table>

Description
This option specifies the way clients wait while initiating connections to servers. The choice of connect strategy depends on the architecture of the client.

The default multithreaded 1f strategy uses the reactor and exhibits non-blocking behavior when initiating a connection to a server. This strategy participates in the leader-follower protocol to synchronize access to the reactor.

The reactive strategy is based on a traditional reactor that waits for events, then loops through event handlers sequentially within a single thread of control.

Finally, in the blocked strategy, the transport-level connector blocks for the duration of the connection attempt with the server. This option should only be used in multithreaded environments.

Usage
In a single-threaded application, either the leader-follower or reactive options are acceptable. In a multithreaded application, either the leader-follower or the blocking options are acceptable.

Impact
The reactive option provides a performance improvement in single threaded applications by avoiding the overhead of thread management and locking.

The blocking option provides the lightest weight code, but pushes the burden of thread management to the application developer.

See Also
17.4.1

Example
static Client_Strategy_Factory "-ORBConnectStrategy reactive"
22.3.2 ORBProfileLock lock_type

Values for lock_type

<table>
<thead>
<tr>
<th>lock_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread (default)</td>
<td>Access to the object profiles is guarded with a mutual exclusion lock.</td>
</tr>
<tr>
<td>null</td>
<td>Object profiles are not locked.</td>
</tr>
</tbody>
</table>

Description
This option controls the locking strategy for the internal state of an object reference. In particular, the profile portion is guarded to avoid modification as the result of receiving LOCATION_FORWARD messages.

By default, a thread mutex lock is used to synchronize access to the profile during forwarding events.

Usage
A client application may safely avoid the use of locks if it is single threaded, if object references are not shared among threads, or if LOCATION_FORWARD messages are guaranteed not to be received.

Impact
Using a null mutex for object references, when it is safe to do so, enhances performance.

The extra lock introduces potential for priority inversion.

Example
static Client_Strategy_Factory "-ORBProfileLock null"
Client Strategy Factory

22.3.3 ORBTransportMuxStrategy strategy

Values for strategy

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exclusive</td>
<td>Only one request may be pending on a connection.</td>
</tr>
<tr>
<td>muxed (default)</td>
<td>Pending requests are multiplexed on a connection.</td>
</tr>
</tbody>
</table>

Description This option controls the request multiplexing strategy of the transport. The exclusive strategy means that no more than one request may be pending on a single connection. If another request is issued, a new connection must be established. The muxed strategy enables more than one pending request on a single connection. With this strategy, pending requests and the related callbacks are stored in a table so that responses may be routed appropriately.

The transport multiplexing strategy is used by the pluggable transport. See 16.4.7 for details of the pluggable transport.

Usage The exclusive strategy is intended for use with ordinary synchronous request/reply behavior. This strategy provides a smaller memory footprint as no table is needed to store pending requests.

Impact The muxed strategy is required for asynchronous messaging. The muxed strategy may also be desirable in a multithreaded client where many client threads are simultaneously invoking synchronous requests through a single ORB to the same target server. If the exclusive strategy is used in this case, multiple connections may be created between the client and server ORBs to transmit multiple simultaneous requests, which can be a problem in environments where resources, such as file descriptors, are scarce.

See Also 17.4.2

Example static Client_Strategy_Factory "-ORBTransportMuxStrategy exclusive"
22.3.4 ORBWaitStrategy *handler_type*

Values for *handler_type*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mt</td>
<td>Use the multithreaded wait strategy, which is based on the leader-follower model.</td>
</tr>
<tr>
<td>mt_noupcall</td>
<td>Use the multithreaded wait strategy, which is based on the leader-follower model, but which does not allow client threads to handle nested upcalls while they are waiting for replies from a server.</td>
</tr>
<tr>
<td>st</td>
<td>Use the single-threaded wait strategy, which allows multiple connection handlers to share a reactor.</td>
</tr>
<tr>
<td>rw</td>
<td>Use the receive-wait wait strategy. Client threads block while waiting for replies from servers.</td>
</tr>
</tbody>
</table>

**Description**

This option specifies the way client threads wait for replies during two-way invocations. In two-way invocations, the client thread is expected to behave synchronously with respect to the server. The client submits a request, then immediately waits for a reply. The choice of wait strategy depends on the architecture of the client application.

The multithreaded strategy implements the *leader-follower* pattern, in which many connection-handler threads are used to asynchronously handle replies from the server.

The single-threaded reactive strategy is based on a traditional reactor that waits for events, then loops through event handlers sequentially within a single thread of control.

In the receive-wait strategy, the connection handler blocks in the *recv()* system call waiting for a reply from the server. This option should only be used in multithreaded environments.

**Usage**

In a single-threaded application, either the multithreaded or single-threaded reactor-based options are acceptable. In a multithreaded application, either the multithreaded or the receive-wait options are acceptable.

**Impact**

The single-threaded reactor option provides a performance improvement over the multithreaded strategy by avoiding the overhead of thread management and locking.

The receive-wait option provides the lightest weight code, but pushes the burden of thread management to the application developer.
The multithreaded-no-upcall option allows multiple connection-handler threads to share the same reactor, but avoids nested upcalls. It is considered an experimental feature.

See Also 17.3.3.2, 20.6.6, 22.3.1, 22.3.3

Example static Client_Strategy_Factory "-ORBWaitStrategy st"

Note If you use the wait-on-read (rw) wait strategy option, you should also use the exclusive transport multiplexing option (22.3.3) to avoid sharing a single connection among threads making concurrent outgoing requests to the same server.

Note You must also use the blocked connect strategy (22.3.1) and the blocking flushing strategy (20.6.6) when using the wait-on-read (rw) wait strategy to keep the client thread from entering the ORB's leader-follower during connection establishment or during output flushing and potentially being used by the ORB to handle incoming requests.
© 2005 Object Computing, Inc.
All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior, written permission of Object Computing, Inc.

Whereas every effort has been made to ensure the technical accuracy of this document, Object Computing, Inc. assumes no responsibility for errors or omissions. This document and technologies described herein are subject to change without notice.

OMG is a registered trademark and Object Request Broker, ORB, OMG IDL, and CORBA are trademarks of Object Management Group, Inc.

Sun, Sun Microsystems, Solaris, Sun Workshop Compiler, Forte, UltraSPARC, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark of The Open Group.

SGI and IRIX are registered trademarks of Silicon Graphics, Inc.

AIX is a registered trademark of IBM.

HP-UX and Tru64 are registered trademarks of Hewlett-Packard Company.


Borland C++ Builder is a registered trademark of Borland Software Corporation.

VxWorks and Tornado are registered trademarks of WindRiver Systems.

LynxOS is a registered trademark of LynuxWorks.

OS-9 is the registered trademark of RadiSys Corporation.

All other products or services noted in this guide are covered by the registered trademarks, service marks, or product names of their respective holders.
Your feedback on, or your submission of content to, this documentation is appreciated. Please contact us at either office location via phone or fax, or e-mail your feedback to: techpubs@ociweb.com.

**How to contact us:**

Object Computing, Inc. (Corporate Office)  
12140 Woodcrest Executive Drive, Suite 250  
St. Louis, MO 63141  
+1.314.579.0066 Voice  
+1.314.579.0065 Fax

Object Computing, Inc.  
64 E Broadway Road, Suite 160  
Tempe, AZ 85282  
+1.480.752.0042 Voice  
+1.480.752.0076 Fax

Support: support@ociweb.com  
Training: training@ociweb.com  
Sales: sales@ociweb.com  
Internet: www.ociweb.com
Part 4

TAO Services
23.1 Introduction

The OMG’s CORBA services specifications define basic services—such as Naming, Events, and Notification—that can be used in a wide variety of applications. Each CORBA service specification includes standard IDL interfaces that help insulate application developers from differences in implementation. Applications benefit by reusing existing components; CORBA vendors benefit by being able to focus on quality of implementation rather than on interface design.

A governing design principle for the OMG services is that each service does only one thing, but it does it very well. For example, the Naming Service and Object Trading Service provide similar functionality. The Naming Service allows applications to look up objects by name; it is like the telephone white pages. By contrast, the Object Trading Service looks up objects based on the properties of the services they provide; it is like the yellow pages. Yet, in spite of the commonality between the object lookup strategies for white pages and yellow pages, there is a separate service defined for each. This minimizes the interfaces and keeps them as simple as possible. It also makes it possible to
use the basic OMG services as building blocks for more sophisticated custom services.

Another governing design principle is to use CORBA features, such as IDL, for specifying the interface separately from the implementation. Consequently, the OMG services from any CORBA compliant ORB can be used by servers and clients from another CORBA compliant ORB. You could, for example, use TAO’s Naming Service for distributed objects that are using a CORBA ORB based on Java, such as JacORB.

As mentioned in Chapter 1, TAO implements a number of the OMG’s CORBA services, as well as certain TAO-specific services. The OMG-defined CORBA services are easily distinguished from non-OMG-defined services by the names of the modules; all the OMG services include the prefix “Cos” in the module name. For example, the Naming Service interfaces are all contained within the module `CosNaming`. Table 23-1 lists the services that are provided with TAO and the libraries with which applications that use each service must link.

### 23.2 Customizing Access to the Services

The CORBA services specifications do not deal with implementation issues such as executable file names or allowable command-line arguments. In fact, you can view the driver programs (the code that initializes and registers the servants implementing the CORBA services) used for the TAO services as sample drivers. If you examine the source code of the drivers you will find that they are quite small and simple; most of the work is done by the servants in the libraries that implement the IDL interfaces of the services.

---

**Note**  
To avoid confusion, when discussing CORBA and TAO services, the term **service** refers to the library code that implements the IDL interfaces, whereas the term **server** refers to the executable that instantiates and initializes the servant(s) running this code.

---

The servants that implement the CORBA and TAO services have been designed so that they can be easily created and initialized within your own application programs. By accessing these services programmatically, you can guarantee that the service you need will be available during the lifetime of
your application, take advantage of collocation optimizations, and reduce the number of processes in your application.

### 23.3 TAO’s ORB Services Libraries

The code that implements the IDL interfaces for each of the services included with TAO is located within one or more libraries. Services with a single library put all their code in that single library which must be linked by all clients and servers participating in that service. Many other services (increasingly with TAO 1.4a) break the service functionality into multiple libraries. A common pattern, shared by several services, is to have individual stub, skeleton, and server libraries. For example, the Naming Service has the following libraries:

- `TAO_CosNaming` - Naming service stub and client code
- `TAO_CosNaming_Skel` - Naming service skeleton code
- `TAO_CosNaming_Serv` - Naming service server code

The `TAO_CosNaming` library contains the code needed by Naming Service clients that only need to interact with this service as CORBA clients. The `TAO_CosNaming_Skel` library contains the generated skeletons and is required if your Naming Service client implements any service-defined IDL interfaces. The `TAO_CosNaming_Serv` library contains all the code necessary in building servers for the Naming Service. The server library is usually includes the vast majority of a service’s code and means that clients no longer need to link this code with their application.

Some services place some of their functionality in additional libraries and require application developers to link those libraries when those features are desired. These feature-oriented libraries are discussed in the chapter documenting that service.

The source code for these libraries is contained in the `$TAO_ROOT/orbsvcs/orbsvcs` directory and its subdirectories. The TAO services and their individual libraries are listed in Table 23-1.

**Table 23-1 TAO ORB Services Libraries**

<table>
<thead>
<tr>
<th>Service</th>
<th>Library Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio/Video Streaming</td>
<td>TAO_AV</td>
</tr>
</tbody>
</table>
## TAO Services Overview

### Table 23-1 TAO ORB Services Libraries

<table>
<thead>
<tr>
<th>Service</th>
<th>Library Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency Control</td>
<td>TAO_CosConcurrency</td>
</tr>
<tr>
<td>Event</td>
<td>TAO_CosEvent</td>
</tr>
<tr>
<td></td>
<td>TAO_CosEvent_Skel</td>
</tr>
<tr>
<td></td>
<td>TAO_CosEvent_Serv</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>TAO_FaultTolerance</td>
</tr>
<tr>
<td></td>
<td>TAO_FT_ClientORB</td>
</tr>
<tr>
<td></td>
<td>TAO_FtorB_Utils</td>
</tr>
<tr>
<td></td>
<td>TAO_FtRtClientORB</td>
</tr>
<tr>
<td></td>
<td>TAO_FtRtEvent</td>
</tr>
<tr>
<td></td>
<td>TAO_FTRTEventChannel</td>
</tr>
<tr>
<td></td>
<td>TAO_FT_ServerORB</td>
</tr>
<tr>
<td>Interface Repository</td>
<td>TAO_IFRService</td>
</tr>
<tr>
<td>Life Cycle</td>
<td>TAO_CosLifeCycle</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>TAO_CosLoadBalancing</td>
</tr>
<tr>
<td>Log (Admin)</td>
<td>TAO_DsLogAdmin</td>
</tr>
<tr>
<td></td>
<td>TAO_DsLogAdmin_Skel</td>
</tr>
<tr>
<td></td>
<td>TAO_DsLogAdmin_Serv</td>
</tr>
<tr>
<td>Log (Event-based)</td>
<td>TAO_DsEventLogAdmin</td>
</tr>
<tr>
<td></td>
<td>TAO_DsEventLogAdmin_Skel</td>
</tr>
<tr>
<td></td>
<td>TAO_DsEventLogAdmin_Serv</td>
</tr>
<tr>
<td>Log (Notify-based)</td>
<td>TAO_DsNotifyLogAdmin</td>
</tr>
<tr>
<td></td>
<td>TAO_DsNotifyLogAdmin_Skel</td>
</tr>
<tr>
<td></td>
<td>TAO_DsNotifyLogAdmin_Serv</td>
</tr>
<tr>
<td>Log (RTEvent-based)</td>
<td>TAO_RTEventLogAdmin</td>
</tr>
<tr>
<td>Naming</td>
<td>TAO_CosNaming</td>
</tr>
<tr>
<td></td>
<td>TAO_CosNaming_Skel</td>
</tr>
<tr>
<td></td>
<td>TAO_CosNaming_Serv</td>
</tr>
<tr>
<td>Notification</td>
<td>TAO_CosNotification</td>
</tr>
<tr>
<td></td>
<td>TAO_CosNotification_Skel</td>
</tr>
<tr>
<td></td>
<td>TAO_CosNotification_Serv</td>
</tr>
<tr>
<td></td>
<td>TAO_CosNotification_Persist</td>
</tr>
<tr>
<td></td>
<td>TAO_RTNotification</td>
</tr>
<tr>
<td>Object Trading</td>
<td>TAO_CosTrading</td>
</tr>
<tr>
<td></td>
<td>TAO_CosTrading_Skel</td>
</tr>
<tr>
<td></td>
<td>TAO_CosTrading_Serv</td>
</tr>
<tr>
<td>“Old” Real-Time Event</td>
<td>TAO_RTOLDEvent</td>
</tr>
<tr>
<td>Property</td>
<td>TAO_CosProperty</td>
</tr>
<tr>
<td>Real-Time Event</td>
<td>TAO_RTEvent</td>
</tr>
<tr>
<td></td>
<td>TAO_RTCORBAEvent</td>
</tr>
<tr>
<td></td>
<td>TAO_RTCosScheduling</td>
</tr>
<tr>
<td></td>
<td>TAO_RTKokyuEvent</td>
</tr>
<tr>
<td></td>
<td>TAO_RTSchedEvent</td>
</tr>
<tr>
<td>Real-Time Scheduling</td>
<td>TAO_RTsched</td>
</tr>
<tr>
<td>Security</td>
<td>TAO_Security</td>
</tr>
<tr>
<td>Time</td>
<td>TAO_CosTime</td>
</tr>
</tbody>
</table>
In addition, some common functionality shared by multiple services is located in utility libraries such as TAO_Svc_Utils, TAO_ETCL, and TAO_PortableGroup.

The combination of multi-library services, dependencies between services, and the utility libraries make generating the full set of service libraries a challenge. The easiest way to resolve these dependencies is to use MPC and TAO’s service base projects to generate your link commands. For example, to build an executable that uses TAO’s Naming Service as a pure CORBA client, create an MPC project that inherits from the namingexe base project. The following sample MPC project file is taken from $TAO_ROOT/orbsvcs/tests/Simple_Naming/Simple_Naming.mpc:

```plaintext
project(*Client) : namingexe, portable_server {
    Source_Files {
        client.cpp
    }
}

The namingexe project simply inherits from the naming and taoexe projects. The naming project ensures that we get the proper include paths, environment variables, and link to the TAO_CosNaming library. You can also derive your projects directly from naming, naming_skel, and naming_serv base projects that TAO defines.

Using any of these projects simplifies your build process and may make it more portable if there are further library changes in later versions of TAO. Look in $ACE_ROOT/bin/MakeProjectCreator/config for all of TAO’s base projects. See Chapter 3 and Chapter 4 for more information on using MPC.

To access the header files for the services, use the form:

```c
#include <orbsvcs/service-nameC.h>
```

For example, to include the header file for the Naming Service, add the following line to your code:

```c
#include <orbsvcs/CosNamingC.h>
```
23.4 Locating Service Objects

Many of TAO’s services can be located by passing a pre-defined object identifier to the ORB’s resolve_initial_references() operation. For example:

```c++
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Resolve the Naming Service.
CORBA::Object_var naming_obj = orb->resolve_initial_references("NameService");
```

When resolve_initial_references() is passed an object identifier (such as "NameService"), it attempts to find the corresponding service object. The resolve_initial_references() implementation performs the following steps, in order, until the operation succeeds, an exception occurs, or the operation times out:

---

**Note**  
The general algorithm here is compliant with section 4.5.3, “Configuring Initial Service References,” of the OMG CORBA 3.0 specification. The details of the environment variables (step 3) and multicast discovery (step 4) are specific to TAO.

---

1. If an initial reference to the service object was previously specified using CORBA::ORB::register_initial_reference(), that object reference is returned.

2. If the initial reference to the service object was specified using the ORBInitRef option, e.g.,

   ```
   -ORBInitRef NameService=corbaloc::tango:2809/NameService
   ```

   then the stringified object reference provided as an argument to -ORBInitRef is passed to CORBA::ORB::string_to_object() to obtain the object reference to the service object. See 19.8.13 for more information about the ORBInitRef option.

3. An environment variable name is constructed by appending “IOR” to the object identifier. If this environment variable is set, its value is passed to CORBA::ORB::string_to_object() as above. For example, when the
Locating Service Objects

4. If the service object identifier is one of NameService, TradingService, InterfaceRepository, or ImplRepoService, then TAO attempts multicast discovery of the service. See 23.4.1 for details of multicast discovery.

5. If the ORBDefaultInitRef option was used, e.g.,

   -ORBDefaultInitRef corbaloc::tango:2809

   then a slash character ("/") plus the service object identifier are appended to the URL prefix provided as an argument to -ORBDefaultInitRef. The resulting stringified object reference is then passed to CORBA::ORB::string_to_object() as above. See 19.8.7 for more information about the ORBDefaultInitRef option.


23.4.1 Multicast Service Discovery

TAO implements a multicast discovery mechanism for locating the following services:

- Naming
- Trading
- Implementation Repository
- Interface Repository

Whenever the CORBA-standard and environment variable mechanisms are not available for locating one of these services, TAO tries to use its multicast mechanism to locate the service. The basic process is that a datagram is sent to a multicast address and port requesting the location of the service. Any servers listening on this address reply with their object reference. If multiple services reply, the client of the service uses the first response it receives. If no server replies within the timeout specified by the preprocessor macro TAO_DEFAULT_SERVICE_RESOLUTION_TIMEOUT defined in $TAO_ROOT/tao/orbconf.h, an exception of type CORBA::ORB::InvalidName is raised.
By default, multicast discovery is enabled for clients, meaning they attempt to use it when other mechanisms are not used. Most servers disable it by default and supply command line options to enable it (typically `-m 1`).

Although multicast discovery works well for some applications, its use is problematic for others. The most common issue encountered is when multiple servers are configured to use the same multicast address. Different client applications then randomly connect to one of the available servers based on which reply is received. Typically, this is not what the application requires or expects.

The following steps describe how the multicast address and port are determined for a given service. Once the address and port are defined the remaining steps are skipped.

1. If the object identifier is `NameService` and the `ORBMulticastDiscoveryEndpoint` option was specified, the value of its argument is used as the IP address and port number.

2. Except for the Interface Repository, each of the other services that use multicast discovery defines an ORB initialization option that allows for run-time specification of the multicast discovery port.

3. The multicast discovery port can also be specified via an environment variable for each of the services.

4. If all of the previous methods fail, the default multicast discovery port number for that service is used. The default port numbers are defined in `$TAO_ROOT/tao/default_ports.h` using preprocessor macros. For example, the Naming Service’s default port number is defined by `TAO_DEFAULT_NAME_SERVER_REQUEST_PORT`.

If the address is not specified in step 1 above, then the default multicast address of 224.9.9.2 is used. The following table summarizes the Object ID, ORB initialization option, environment variable, and default port for each of the services that use multicast.

### Table 23-2 Multicast Discovery Services

<table>
<thead>
<tr>
<th>Object ID</th>
<th>ORB Initialization Option</th>
<th>Environment Variable</th>
<th>Default Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>NameService</td>
<td>-ORBNameServicePort</td>
<td>NameServicePort</td>
<td>10013</td>
</tr>
<tr>
<td>TradingService</td>
<td>-ORBTradingServicePort</td>
<td>TradingServicePort</td>
<td>10016</td>
</tr>
<tr>
<td>ImplRepoService</td>
<td>-ORBImplRepoServicePort</td>
<td>ImplRepoServicePort</td>
<td>10018</td>
</tr>
<tr>
<td>InterfaceRepository</td>
<td>N/A</td>
<td>InterfaceRepoServicePort</td>
<td>10020</td>
</tr>
</tbody>
</table>
In addition, multicast discovery can also be specified via the -ORBInitRef and -ORBDefaultInitRef options. See 19.8.13 for more details.
Chapter 24

Naming Service

24.1 Introduction

The OMG Naming Service version 1.2 (OMG Document formal/02-09-02) defines a means for mapping names to object references. Names are composed of sequences of simple structures that are, in turn, constructed from text strings. The Naming Service allows applications in a distributed system to locate CORBA objects by name rather than by caching object references or passing around stringified IORs. It is similar to the white pages in a phone book that maps telephone numbers to names. Chapter 18 of Advanced CORBA Programming with C++ presents a detailed discussion of the Naming Service.

TAO provides a complete implementation of the Naming Service specification. The Naming Service may be run as a stand-alone executable called Naming_Service, or it can be built into an application by linking in the appropriate libraries. The Naming Service interfaces and data types from the OMG’s CosNaming module are defined in the file $TAO_ROOT/orbsvcs/orbsvcs/CosNaming.idl. Applications that use TAO’s Naming Service implementation should include the header file $TAO_ROOT/orbsvcs/orbsvcs/CosNamingC.h and link with the
TAO_CosNaming library. Applications that wish to implement the Naming Service internally will need to link with the TAO_CosNaming_Skel and TAO_CosNaming_Serv libraries.

24.1.1 Road Map
The Naming Service is perhaps the most widely used of all the OMG CORBA services. It is well documented in sources such as Advanced CORBA Programming with C++ and Pure CORBA. Therefore, this chapter does not attempt to completely explain how to use the standard Naming Service from your application. Instead, this chapter focuses on the TAO-specific features of the Naming Service.

If you want to learn more about...

- how to find the Naming Service from within your application, see 24.2, “Resolving the Naming Service.”
- code that uses the basic Naming Service features from an application’s clients and servers, see 24.3, “Naming Service Example.”
- using the “corbaloc” and “corbaname” Object URL features of the Interoperable Naming Service specification, see 24.4, “Object URLs.”
- code that uses the extended Naming Service interfaces (NamingContextExt), see 24.5, “The NamingContextExt Interface.”
- TAO’s classes that implement Naming Service features, for both clients and servers, see 24.6, “TAO-Specific Naming Service Classes.”
- how to use TAO’s Naming Service utility programs, see 24.7, “Naming Service Utilities.”
- command-line options affecting TAO’s Naming_Service executable, including the Naming Service persistence features, see 24.8, “Naming Service Command Line Options.”

To fully understand and take advantage of the Naming Service’s features, be sure to read Chapter 18 of Advanced CORBA Programming with C++ or Chapter 6 of Pure CORBA in addition to this chapter.

Full source code for all the examples presented in this chapter is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/NamingService.
24.2 Resolving the Naming Service

To find the Naming Service, your application can use the CORBA::ORB::resolve_initial_references() operation, passing it the string “NameService”. For example:

// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Resolve the Naming Service.
CORBA::Object_var naming_obj = orb->resolve_initial_references("NameService");

CORBA::ORB::string_to_object() may also be used to resolve the Naming Service. Examples using this technique are provided in 23.4.

When resolve_initial_references() is passed the string “NameService”, it attempts to find the root Naming Context of the Naming Service. TAO’s resolve_initial_references() implementation is discussed in more detail in 23.4.

If resolve_initial_references() succeeds, it returns the CORBA object reference to the Naming Service's root Naming Context. To use this object reference as a CosNaming::NamingContext, you must first narrow it, as follows:

CosNaming::NamingContext_var root =
   CosNaming::NamingContext::_narrow(naming_obj.in());

TAO’s Naming Service implements the CosNaming::NamingContextExt interface from OMG Document formal/02-09-02, so you can also narrow the object reference returned from resolve_initial_references() as follows:

CosNaming::NamingContextExt_var root =
   CosNaming::NamingContextExt::_narrow(naming_obj.in());
24.3 Naming Service Example

To help you become familiar with the TAO Naming Service, we now present a simple example based on the Messenger example which was introduced in Chapter 3.

Recall that the original Messenger server writes an IOR as a string to a file. The client then reads the IOR string from the file and converts it back into an object reference. Of course, to use this mechanism, the client and server must be running on the same machine or at least on machines that share a file system, or you must transmit the stringified IOR to the client through some other means. To overcome this limitation, we will use the Naming Service and assume that both the server and client have access to the Naming Service.

24.3.1 Source Code Listings for the Example

We have modified the Messenger server and client source code to use the Naming Service rather than working with a stringified IOR. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/NamingService/Messenger.

24.3.1.1 Server C++ Source Code File

The server now uses the Naming Service rather than writing its IOR as a string to a file. The implementation of the server is in MessengerServer.cpp.

```cpp
#include "Messenger_i.h"
#include <orbsvcs/CosNamingC.h>
#include <iostream>

int main(int argc, char* argv[]) {
    try {
        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Get reference to Root POA
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

        // Activate POA Manager
        PortableServer::POAManager_var mgr = poa->the_POAManager();
        mgr->activate();
    }

    return 0;
}
```
// Create an object
Messenger_i messenger_servant;

// Find the Naming Service
CORBA::Object_var naming_obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var root = 
    CosNaming::NamingContext::narrow(naming_obj.in());
if(CORBA::is_nil(root.in())) {
    std::cerr << "Nil Naming Context reference" << std::endl;
    return 1;
}

// Bind the example Naming Context, if necessary
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("example");
try {
    CORBA::Object_var dummy = root->resolve(name);
} catch (const CosNaming::NamingContext::NotFound &) {
    CosNaming::NamingContext_var dummy = root->bind_new_context(name);
}

// Bind the Messenger object
name.length(2);
name[1].id = CORBA::string_dup("Messenger");

PortableServer::ObjectId_var oid = 
    poa->activate_object(&messenger_servant);
CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
root->rebind(name, messenger_obj.in());

std::cout << "Messenger object bound in Naming Service" << std::endl;

// Accept requests
orb->run();
orb->destroy();
}
catch(CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
}
return 0;
24.3.1.2 Client C++ Source Code File

The client now uses the Naming Service rather than reading the server object’s IOR as a string from a file. The implementation of the client is in the file MessengerClient.cpp.

```cpp
#include "MessengerC.h"
#include <CosNamingC.h>
#include <iostream>

int main(int argc, char* argv[]) {
    try {
        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Find the Naming Service
        CORBA::Object_var naming_obj = 
            orb->resolve_initial_references("NameService");
        CosNaming::NamingContext_var root = 
            CosNaming::NamingContext::_narrow(naming_obj.in());
        if(CORBA::is_nil(root.in())) {
            std::cerr << "Nil Naming Context reference" << std::endl;
            return 1;
        }

        // Resolve the Messenger object
        CosNaming::Name name;
        name.length(2);
        name[0].id = CORBA::string_dup("example");
        name[1].id = CORBA::string_dup("Messenger");
        CORBA::Object_var obj = root->resolve(name);

        // Narrow the Messenger object reference
        Messenger_var messenger = Messenger::_narrow(obj.in());
        if (CORBA::is_nil(messenger.in())) {
            std::cerr << "Not a Messenger reference" << std::endl;
            return 1;
        }

        // Send a message
        CORBA::String_var message = CORBA::string_dup("Hello!");
        messenger->send_message("TAO User", "TAO Test", message.inout());
    }
    catch (CORBA::Exception& ex) {
        std::cerr << "Caught a CORBA exception: " << ex << std::endl;
        return 1;
    }
    std::cout << "Message was sent" << std::endl;
```
24.3 Naming Service Example

24.3.2 Building the Example
The example includes an MPC file Messenger.mpc that defines projects for building MessengerServer and MessengerClient. These projects can be built from the workspace generated from
$TAO_ROOT/DevGuideExamples/DevGuideExamples.mwc. See Chapter 3 and Chapter 4 for information on using MPC.

24.3.3 Running the Example
Now that you have built the server and client, you are ready to run the example using the Naming Service. The first step is to start the Naming Service, which can be done with or without support for multicast discovery.

24.3.3.1 Starting the Naming Service with Multicasting
The TAO Naming Service server executable is typically found in the directory
$TAO_ROOT/orbsvcs/Naming_Service.

TAO clients will try to locate the Naming Service as described in 24.2. If they cannot locate the Naming Service by another method, they will attempt to locate it using multicast discovery. When invoked with the \(-m 1\) option, the Naming Service will listen for these multicast service discovery requests and respond with the IOR of its root naming context.

Multicasting is connectionless, so a process sending a request need not wait for a recipient to respond. However, datagrams do not guarantee message delivery, nor are messages necessarily received in the same order as they were transmitted.

Note For more details see the discussions of multicast service discovery in 23.4.1 and the mcast: address in 19.8.7.

If your network supports multicast, and you want your applications to find the Naming Service using multicast service discovery, start the TAO Naming Service server as follows (assuming that Naming_Service is in your PATH):

```
$ Naming_Service -m 1

Next, run the MessengerServer in its own terminal window:

$ ./MessengerServer

and run the MessengerClient from a different terminal window:

$ ./MessengerClient

You should see the following messages in the MessengerServer’s window:

Message from: TAO User
Subject:      TAO Test
Message:      Hello!

and in the MessengerClient’s window:

message was sent

The client terminates, but the MessengerServer will continue to run until you kill it. After you kill the MessengerServer, the Naming Service will have a binding to an object reference that is no longer valid (persistent object references, notwithstanding). You could kill the Naming Service or send a request to the Naming Service to unbind the reference.

Compare this with the commands needed to run the server and client without the Naming Service as given in 3.3.8. You no longer need to share files between the server and the client. Using the Naming Service makes it possible to manage a large number of objects without relying on the existence of a file system shared among the distributed objects in the system.

**Discriminating between Multiple Naming Services**

Suppose you want to access a particular Naming Service server when another Naming Service server is running and both are using multicast for service discovery. For example, there may be a deployment Naming Service running on your subnetwork, but you want to test your code using a development Naming Service without impacting users of the more stable deployment service.

You could use the -ORBListenEndpoints and -ORBInitRef options as described in 23.4, thus avoiding multicast discovery of the Naming Service.
Or, you could specify a multicast port other than the default Naming Service multicast discovery port, as follows:

- Set the multicast port that the Naming Service server should use. For our example, we will use port 10999:

  $ Naming_Service -m 1 -ORBNameServicePort 10999

- Provide clients of the Naming Service the port to use for locating the Naming Service. Do this by setting the NameServicePort environment variable to 10999 before running applications that access the Naming Service (shown here using UNIX shell syntax):

  $ NameServicePort=10999; export NameServicePort

  or pass the port as a command line option:

  $ ./MessengerServer -ORBNameServicePort 10999 &

You may want to specify a multicast address and a port other than the default Naming Service values. Do this as follows:

- Set the multicast address and port for the Naming Service server to use. For our example, we will use address 224.0.0.3 and port 10999:

  $ Naming_Service -m 1 -ORBMulticastDiscoveryEndpoint 224.0.0.3:10999

- Provide clients of the Naming Service the address and port to use for locating the Naming Service. Do this by passing the address and port as a command line option:

  $ ./MessengerServer -ORBMulticastDiscoveryEndpoint 224.0.0.3:10999

  or

  $ ./MessengerServer -ORBInitRef \ 
  NameService=mcast://224.0.0.3:10999::/NameService

Another technique is to have the Naming Service server write the IOR of the root Naming Context to a file:

  $ $TAO_ROOT/orbsvcs/Naming_Service/Naming_Service -o /tmp/ns.ior&
$ ./MessengerServer -ORBInitRef NameService=file:///tmp/ns.ior

24.3.3.2 Starting the Naming Service without Multicasting

Suppose you do not have the ability to use multicasting for your particular network configuration or you choose not to use multicasting for discovery of the Naming Service. The approach we use in this case is to specify an endpoint on which the Naming Service server’s ORB will listen for CORBA requests, then specify the Naming Service’s object reference to the applications using the \texttt{-ORBInitRef} option.

Run the Naming Service server in a separate session:

\begin{verbatim}
$ Naming_Service -ORBListenEndpoints iiop://tango:2809
\end{verbatim}

The \texttt{-ORBListenEndpoints} option is passed to CORBA::ORB\_init() and causes the Naming Service server’s ORB to listen for requests on the specified endpoint. See 19.8.17 for more information on the \texttt{-ORBListenEndpoints} option.

Run \texttt{MessengerServer} and \texttt{MessengerClient} in another session. We specify the object reference of the Naming Service’s root Naming Context by passing the \texttt{-ORBInitRef} option to each one:

\begin{verbatim}
$ ./MessengerServer \ 
-ORBInitRef NameService=corbaloc:iiop:tango:2809/NameService & \\
$ ./MessengerClient \ 
-ORBInitRef NameService=corbaloc:iiop:tango:2809/NameService
\end{verbatim}

The \texttt{-ORBInitRef} option is passed to CORBA::ORB\_init() and initializes the application’s ORB with the association between the string \texttt{NameService} and the \texttt{corbaloc} Object URL provided as an argument. When the application invokes \texttt{resolve\_initial\_references(\texttt{NameService})}, the Object URL is passed to CORBA::ORB::\texttt{string\_to\_object()} as described in 24.2. See 19.8.13 for more information on the \texttt{-ORBInitRef} option. See 24.4.1 for more information on using \texttt{corbaloc} Object URLs.

You can also specify the Naming Service IOR using the \texttt{NameServiceIOR} environment variable, as shown here using UNIX shell syntax:

\begin{verbatim}
$ NameServiceIOR=corbaloc:iiop:tango:2809/NameService; export NameServiceIOR
\end{verbatim}
24.4 Object URLs

24.4.1 corbaloc URL

An application can find the Naming Service by passing a URL to CORBA::ORB::string_to_object() instead of using CORBA::ORB::resolve_initial_references().

The corbaloc URL syntax provides an easy way for users to manipulate a string form of an object reference. It has the syntax:

"corbaloc:"[<protocol id>]:"<protocol addr>

There are two protocols defined in the specification:

- The “resolve initial references” form:

  "corbaloc:rir:"[/<keystring>].

  For example: corbaloc:rir:/NameService.

- The IIOP form:

  "corbaloc:"["iiop"]:"<host>"[":<port>][/<keystring>].

  For example: corbaloc:iiop:rome.here.com:2204.

The default corbaloc protocol is IIOP with a default port of 2809, as defined in the CORBA Core specification. Therefore corbaloc::rome.here.com is equivalent to corbaloc:iiop:rome.here.com:2809.

The corbaloc syntax allows for future/proprietary protocols. TAO supports the proprietary protocols uiop and shmiop.

The corbaloc syntax may be used for the address specified in the options -ORBDefaultInitRef and -ORBInitRef, but not for -ORBListenEndpoints. The corbaloc:rir protocol can be used for neither -ORBDefaultInitRef nor -ORBInitRef because circular references would result.
Naming Service

The description given above is simplified, but covers the most frequently utilized parts of the corbaloc syntax. See 19.8.7 and 19.8.13 for more information on the protocols supported by TAO. See the CORBA 3.0.3 specification, 13.6.10.1, for the full description of the corbaloc URL syntax and semantics.

24.4.1.1.corbaloc Example Code

We have modified the NamingService/Messenger client source code to use string_to_object(). Full source code for this example is in the TAO source code distribution in the directory $TAO_ROOT/DevGuideExamples/NamingService/corbaloc_Messenger. Only MessengerClient.cpp has changed. The MessengerClient now accepts a command line parameter that is a URL to be passed to string_to_object(). The only change is that the call to resolve_initial_references() has been replaced with:

```cpp
char* url = "corbaloc:rir:/NameService"; // default URL to InitRef
if (argc < 2) {
    std::cout << "Defaulting URL to " << url << std::endl;
    std::cout << "Usage: " << argv[0] << " [-ORB options] [corbaloc URL for the name service]"
    << std::endl;
}
else {
    url = argv[1];
}

// Find the Naming Service
CORBA::Object_var naming_obj = orb->string_to_object(url);
```

The new MessengerClient will use a URL provided as a command line parameter to locate the Naming Service.

24.4.1.2. Running the corbaloc example

Here are some examples of using the corbaloc object URL with the updated MessengerClient example.

Start the Naming Service server using IIOP on the local machine:

```bash
Naming_Service -ORBListenEndpoints iiop://localhost:2809
```
24.4 Object URLs

Start the MessageServer using -ORBDefaultInitRef with a corbaloc URL (recall that the default port for corbaloc is 2809 and the default transport is IIOP):

```
MessengerServer -ORBDefaultInitRef corbaloc::localhost
```

Or start the MessageServer using -ORBInitRef with a corbaloc URL:

```
MessengerServer -ORBInitRef NameService=corbaloc::localhost/NameService
```

Or specify a particular port:

```
MessengerServer -ORBInitRef NameService=corbaloc::localhost:2809/NameService
```

Now, start the MessengerClient using the URL parameter (default to port 2809):

```
MessengerClient corbaloc::localhost/NameService
```

Or start the MessengerClient using a corbaloc:rir form URL (you must specify the initial reference):

```
MessengerClient -ORBInitRef NameService=corbaloc::localhost:2809/NameService corbaloc:rir:/NameService
```

24.4.2 corbaname

An application can find both the Naming Service and a name in the Naming Service by passing a corbaname URL to string_to_object().

The corbaname URL syntax is

```
"corbaname:"<corbaloc> ['#'<string_name>]
```

where <corbaloc> is the address of the Naming Service and 
<string_name> is the stringified name of the object to be found in the Naming Service. A stringified name is a string representation of the 
CosNaming::Name sequence with the syntax:

```
<string_name> = <name_component>['/'<name_component>]*
<name_component> = <name>['.'<type>]
```

Here are some examples:
corbaname:rir:#root/middle.my_type/leaf
corbaname::ns_node:9999#usa/arizona/tempe

The backslash ‘\’ character escapes the reserved meaning of ‘/’, ‘.’, and ‘\’ in a
stringified name.

corbaname has an additional escaping requirement for the ‘\’ backslash
character and other special characters. If a character that requires escaping is
present in a name component it is encoded as two hexadecimal digits
following a ‘%’ character to represent the octet.

A CosNaming::Name “leaf/esc_slash” and type “leaf_type” under the name
“root.esc_dot” would have the stringified name of:

\root.esc_dot/leaf/esc_slash.leaf_type

And escaped for corbaname thus:

root%5c.esc_dot/leaf%5c/esc_slash.leaf_type

See the to_string() and to_url() method descriptions in 24.5 for another
example of escaping.

24.4.2.1 corbaname Example Code

We have modified the NamingService/Messenger client source code to
pass a corbaname URL to string_to_object(). Full source code for this
example is in the TAO 1.4a source code distribution in the directory
$TAO_ROOT/DevGuideExamples/NamingService/corbaname_Messen
ger. Only MessengerClient.cpp has changed. The MessengerClient
now accepts a command line parameter that is a URL to be passed to
string_to_object(). The only change is that the following code:

// Find the Naming Service
CORBA::Object_var naming_obj = orb->resolve_initial_references("NameService");

1. US-ASCII alphanumeric characters and the following: ":" | "/" | ":" | ":" | ":" | ":" | "@" |
"&" | ":" | "$" | ":" | ":" | ":" | ":" | "!" | ":" | "#" | "" | "("")" are not escaped.
All others must be escaped.

2. common escapes: ‘\’ => "%5c", ‘<’ => "%3e", ‘>’ => "%3e", ‘ ’ => "%20", ‘%’ =>
"%25"
CosNaming::NamingContext_var root =
CosNaming::NamingContext::_narrow(naming_obj.in());
Naming Service

```cpp
if (CORBA::is_nil(root.in())) {
    std::cerr << "Nil Naming Context reference" << std::endl;
    return 1;
}

// Resolve the Messenger object
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("example");
name[1].id = CORBA::string_dup("Messenger");
CORBA::Object_var obj = root->resolve(name);

has been replaced with:

```cpp
char* url = "corbaname:rir:#example/Messenger"; // default URL to InitRef
if (argc < 2) {
    std::cout << "Defaulting URL to " << url << std::endl;
    std::cout << "Usage: " << argv[0]
    << " [-ORB options] [corbaname URL for message server]" << std::endl;
} else {
    url = argv[1];
}

// Resolve the Naming Service and the Messenger.
CORBA::Object_var obj = orb->string_to_object(url);
```

The last statement first finds the Naming Service, then resolves the object reference of the Messenger using the name "example/Messenger" (relative to the root Naming Context).

### 24.4.2.2 Running the corbaname example

Start the Naming Service server and MessengerServer as described in 24.4.1.2.

Start the client with a corbaname that does not require the -ORBDefaultInitRef option:

```shell
MessengerClient corbaname:iiop:localhost:2809#example/Messenger
```

Or defaults to use the IIOP protocol:

```shell
MessengerClient corbaname::localhost:2809#example/Messenger
```

Or default protocol (IIOP) and default port (2809):
24.5 The NamingContextExt Interface

The NamingContextExt interface is derived from NamingContext and defines operations to convert between CosNaming::Names and stringified names. The interface is defined in $TAO_ROOT/orbsvcs/orbsvcs/CosNaming.idl as follows:

```idl
module CosNaming
{
    // ... other interfaces ...

    interface NamingContextExt : NamingContext
    {
        typedef string StringName; // Stringified form of a Name.
        typedef string Address;    // URL<address> such as myhost.xyz.com.
        typedef string URLString;  // Stringified form of a URL<address>.
        StringName to_string (in Name n) raises (InvalidName);
        Name to_name (in StringName sn) raises (InvalidName);
        exception InvalidAddress {};
        URLString to_url (in Address addr,
                           in StringName sn)
                         raises (InvalidAddress, InvalidName);
        Object resolve_str (in StringName n)
                         raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
    }
};
```

24.5.1 NamingContextExt Operations

The operations of the NamingContextExt interface are:

- `to_string()`: Converts a CosNaming::Name to a stringified name. If the Name is invalid, an exception of type CosNaming::NamingContext::InvalidName is raised.
• **to_name()**: Converts a stringified name to a `CosNaming::Name`. If the stringified name is syntactically malformed or violates an implementation limit, an exception of type `CosNaming::NamingContext::InvalidName` is raised.

• **resolve_str()**: A convenience operation that resolves a name in the same manner as `resolve()`, but it accepts a stringified name as a parameter instead of a `CosNaming::Name`.

• **to_url()**: Converts a corbaloc URL `<address>/<key_string>` component and a stringified name to a fully formed corbaloc Object URL. Examples of the `<address>/<keystring>` parameter are:
  - `iiop:rome.phx.ociweb.com`
  - `:rome.phx.ociweb.com/a/b/c`
  - `shmiop:12345/a/b/c`

`to_url()` performs any escapes necessary on the parameters and returns a fully formed URL string. An exception is raised if either the corbaloc address and key parameter or name parameter are malformed. It is illegal for the stringified name to be empty. If the address is empty, an exception of type `CosNaming::NamingContextExt::InvalidAddress` is raised.

### 24.5.2 Name Conversion Examples

Here are some examples using the `NamingContextExt` interface:

1. The following code shows the initialization of a `CosNaming::NamingContextExt` object and a call to the `to_string()` method:

   ```cpp
   // Find the Naming Service
   CORBA::Object_var naming_obj = 
   orb->resolve_initial_references("NameService");
   CosNaming::NamingContextExt_var naming_context_ext = 
   CosNaming::NamingContextExt::_narrow(naming_obj.in());
   if (CORBA::is_nil(naming_context_ext.in())) {
     std::cerr << "Nil Naming Context reference" << std::endl;
     return 1;
   }
   
   CosNaming::Name name;
   name.length(2);
   name[0].id = CORBA::string_dup("root.esc-dot");
   ```
name[0].kind = CORBA::string_dup("kind1");
name[1].id = CORBA::string_dup("leaf/esc-slash");
name[1].kind = CORBA::string_dup("kind2");

// Convert Name to String Name.
CORBA::String_var str_name = naming_context_ext->to_string(name);
std::cout << "str_name:  " << str_name.in() << std::endl;

It produces the following output:

str_name root/esc-dot.kind1/leaf/esc-slash.kind2

Notice that the "." and "/" characters in the Name have been escaped.

2. The following code converts the str_name() above back into a CosNaming::Name object:

std::cout << "str_name:  " << str_name.in() << std::endl;

// Convert String Name to Name.
CosNaming::Name * tname = naming_context_ext->to_name(str_name);
std::cout << "converted back to a CosNaming::Name: " << std::endl;
std::cout << "   name[0] = " << (* tname)[0].id.in() << " , "
       << (* tname)[0].kind.in() << std::endl;
std::cout << "   name[1] = " << (* tname)[1].id.in() << " , "
       << (* tname)[1].kind.in() << std::endl;

With the following output:

str_name:  root/esc-dot.kind1/leaf/esc-slash.kind2
converted back to a CosNaming::Name:
name[0] = root.esc-dot, kind1
name[1] = leaf/esc-slash, kind2

3. The following code shows the to_url() method being used:

// Create a URL string for the application object.
CORBA::String_var address =
    CORBA::string_dup(":berlin.phx.ociweb.com:2809/key/str");
std::cout << "call to_url("" << address.in() << "")" << std::endl;
std::cout << "CORBA::String_var url_string =
    naming_context_ext->to_url (address.in(), str_name.in());

name[0].kind = CORBA::string_dup("kind1");
name[1].id = CORBA::string_dup("leaf/esc-slash");
name[1].kind = CORBA::string_dup("kind2");

// Convert Name to String Name.
CORBA::String_var str_name = naming_context_ext->to_string(name);
std::cout << "str_name:  " << str_name.in() << std::endl;
Naming Service

std::cout << "to_url result: " << url_string.in() << std::endl;

With these results:

call to_url(":berlin.phx.ociweb.com:2809/key/str"
,"root\.esc-dot.kind1/leaf/esc-slash.kind2")
to_url result:
corbaname::berlin.phx.ociweb.com:2809/key/str#root%5c.esc-dot.kind1/leaf%5c/esc-slash.kind2

24.6 TAO-Specific Naming Service Classes

24.6.1 Using the TAO_Naming_Client Class

TAO defines a TAO_Naming_Client class that simplifies the interface for accessing and using the Naming Service. The resulting code is now dependent on this TAO specific class, but the class would be trivial to port for use with other ORBs if portability is an issue.

The remainder of this section shows the changes necessary to convert the previous example to use the TAO_Naming_Client class. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/NamingService/Naming_Client.

To use the TAO_Naming_Client, you must add the following include directive:

```
#include <orbsvcs/Naming/Naming_Client.h>
```

Now instead of calling resolve_initial_references() and narrowing the resulting object reference, simply create a TAO_Naming_Client object and call init() on it.

```
// Find the Naming Service
TAO_Naming_Client naming_client;

if(naming_client.init (orb.in ()) != 0){
    std::cerr << "Could not initialize naming client."
                  << std::endl;
    return 1;
}
```
The `init()` member function will return a non-zero value when it fails to locate the Naming Service and initialize properly. Now, the naming client can be used as a smart pointer to access the root Naming Context via `operator->()`. For example, in the `MessengerClient`:

```cpp
CORBA::Object_var obj = naming_client->resolve(name);
```

Similarly, in the `MessengerServer`:

```cpp
naming_client->rebind(name, messenger_obj.in());
```

### 24.6.2 Using the TAO_Naming_Server Class

When used with the `TAO_Naming_Server` class and threads, the Naming Service can easily be collocated within any application. This may be desirable for many reasons, for example:

- When an embedded target system (e.g., VxWorks) does not support processes, requiring the Naming Service to run in a separate thread.
- For performance reasons you may want the Naming Service collocated to avoid network traffic.
- You may want the Naming Service collocated with other code that handles graceful shutdown.

The remainder of this section shows the changes necessary to convert the previous example to use the `TAO_Naming_Server` class. Full source code for this example is in the TAO 1.4a source code distribution in the directory `$TAO_ROOT/DevGuideExamples/NamingService/Naming_Server`.

### 24.6.3 Example using TAO_Naming Classes

The `ACE_Task_Base` base class, defined in `$ACE_ROOT/ace/Task.h`, is used in this example to create individual threads for running the Naming Service and a `MessengerServer` (as in the previous example).

#### 24.6.3.1 Source Code

Each class overrides the base class function `svc()`, which is the entry point for both threads as they are activated. Both tasks create and initialize their own ORB, because calling `run()` on the same ORB has undesirable effects. The declaration of the Naming Service task is in the file `NamingTask.h`.
#include <ace/Task.h>

class NamingTask : public ACE_Task_Base
{
public:
    NamingTask (int argc, char* argv[]);
    virtual int svc();

private:
    int argc_;  
    char** argv_;  
};

The implementation of the Naming Service task is in the file NamingTask.cpp.

#include "NamingTask.h"
#include <orbsvcs/Naming/Naming_Server.h>

NamingTask::NamingTask (int argc, char* argv[])
: argc_ (argc),
  argv_ (argv)
{}

int NamingTask::svc()
{
    int status = -1;

    try {
        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc_, argv_, "NamingORB");

        // Get reference to Root POA
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

        // Activate POA Manager
        PortableServer::POAManager_var poaManager = poa->the_POAManager();
        poaManager->activate();

        // Initialize the Naming Service
        // We are not going to look for other naming servers
        TAO_Naming_Server naming;
        if (naming.init(orb.in(),
                        poa.in(),
                        ACE_DEFAULT_MAP_SIZE,
                        0,}
0) == 0) {
  std::cout << "The Naming Service Task is ready." << std::endl;

  // Accept requests
  orb->run();
  status = 0;
} else {
  std::cerr << "Unable to initialize the Naming Service." << std::endl;
}
} catch (CORBA::Exception& ex) {
  std::cerr << "CORBA exception: " << ex << std::endl;
}

return status;

The implementation of the Messenger server task, located in the file
MessengerTask.cpp, looks very similar to the main() function of the
Messenger Server in the previous example, so to avoid repetition, we will not
list it here.

The main() function for this example, which uses the NamingTask and
MessengerTask, is listed in the file NamingMessenger.cpp.

#include "NamingTask.h"
#include "MessengerTask.h"
#include <ace/OS.h>

int main(int argc, char* argv[]) {

  // Start the Naming Service task
  NamingTask namingService(argc, argv);
  namingService.activate();

  // Wait for it to initialize
  ACE_OS::sleep(5);

  // Start the Messenger task
  MessengerTask messenger;
  messenger.activate();

  // Wait for all tasks to complete
  namingService.thr_mgr() -> wait();

  return 0;
}
24.6.3.2 Running the Example

Assuming that the MessengerClient (from the previous example) already has been built, you now are ready to run this example. Begin by starting the NamingMessenger binary. This will create the Naming Service thread and then the Messenger Server thread. After the ready statements for both are printed, run the MessengerClient to see the interaction between the Naming Service and the MessengerServer.

24.7 Naming Service Utilities

Several utilities for managing and debugging TAO applications and the Naming Service come with TAO (located in $TAO_ROOT/utils).

24.7.1 nslist, nsadd and nsdel utilities

nslist is useful for listing the current Naming Service bindings. nsadd and nsdel are limited to the root naming context (and are therefore of limited utility).

- **nslist**: list the bindings in the Naming Service

  Usage: nslist [ [--ior][--ctxior] | --nsior ]
  -- = display the usage
  --ior = output the IOR of each entry instead of short address
  --ctxior = output the IOR of each Naming Context
  --nsior = output the IOR of the root Naming Context

- **nsadd**: add a binding relative to the root NamingContext

  Usage: nsadd --name <name> --ior <ior> [ --rebind ] [--newcontext]
  -- = display the usage
  --name <name> = the name of the entry to add
  --ior <ior> = the IOR to add (can use corbaname, IOR, and other formats)
  --rebind = rebind to name (may overwrites existing binding)
  --newcontext = creates a new Naming Context and binds it to name

- **nsdel**: delete a binding relative to the root NamingContext

  Usage: nsdel [--name <name>] [--destroy]
--name <name> = unbind (remove) the binding specified by name
--destroy = unbind and destroy the context bound to name

24.7.2 NamingViewer utility

NamingViewer is an MFC (Microsoft Foundation Classes) application for viewing and manipulating the bindings in the Naming Service. wxNamingViewer is similar to NamingViewer, but uses wxWindows (a cross-platform GUI toolkit) instead of MFC.

To use NamingViewer, you must select or add a Naming Service to display its tree as shown in Figure 24-1. Naming Services are added to your Windows registry so they are available the next time you start the NamingViewer. The Naming Service IOR should include the ending stringified object key "/NameService", for example:

corbaloc::localhost:2809/NameService
Figure 24-2 shows the naming tree and Messenger binding after running the NamingService/Messenger example. Double clicking on an object/leaf in the tree will open a View IOR dialog (as shown). Clicking using the right-most mouse button on an object/leaf or context node will display a menu of operations for that entry. Objects/leaves may be unbound or viewed. Operations associated with a context include: bind, bind new context, unbind object, unbind, destroy, view reference and refresh.
Figure 24-2 Display Name Service Entry
24.8 Naming Service Command Line Options

As stated previously, the TAO Naming Service server may be run as a stand-alone executable called *Naming Service*, found in the path $TAO_ROOT/orbsvcs/Naming_Service. Table 24-1 lists command-line options for controlling the behavior of the TAO Naming Service server.

**Table 24-1 Naming_Service Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-b</td>
<td>Address to be used for memory mapping the Naming Service state file, identified by the *persistence_file_name_.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Platform dependent.</td>
<td></td>
</tr>
<tr>
<td>-d</td>
<td>Provides Naming Service specific debug information.</td>
<td>No diagnostics given.</td>
</tr>
<tr>
<td>-f persistence_file_name</td>
<td>Specify the name of a file in which to store the Naming Contexts and bindings so they can be read if the Naming Service needs to be restarted.</td>
<td>Do not store Naming Contexts and bindings.</td>
</tr>
<tr>
<td>-m 0</td>
<td>1</td>
<td>Specify whether the Naming Service should listen for multicast requests. If -m 0 is used, the Naming Service does not listen for multicast requests.</td>
</tr>
<tr>
<td>-p pid-file-name</td>
<td>Specify the name of the output file for writing the process ID as a string.</td>
<td>Do not write the process ID.</td>
</tr>
<tr>
<td>-o ior-file-name</td>
<td>Specify the name of the output file for writing the IOR of the Naming Service as a string.</td>
<td>Do not write the IOR.</td>
</tr>
<tr>
<td>-r directory</td>
<td>Use redundant flat-file persistence; same as the -u option, except that more than one instance of the TAO Naming Service server can be run, each using the same set of disk files, to achieve a degree of fault tolerance (as long as directory is accessible to both servers).</td>
<td>No redundant Naming Service server persistence.</td>
</tr>
<tr>
<td>-s context-size</td>
<td>Specify the size of the hash table to allocate when creating Naming Contexts.</td>
<td>1024</td>
</tr>
<tr>
<td>-t listen-time</td>
<td>Specify how many seconds the server should listen for requests before exiting.</td>
<td>Listen indefinitely.</td>
</tr>
<tr>
<td>-u directory</td>
<td>Use a flat-file persistence implementation that stores object reference information in a file per context. Each context file is placed in the directory specified.</td>
<td>Do not store Naming Contexts and bindings.</td>
</tr>
</tbody>
</table>
### 24.8 Naming Service Command Line Options

#### 24.8.1 Using the Naming Service Persistence Options

There are three options that provide persistence of the relationships between names and object references: \(-f\), \(-u\), and \(-r\). The \(-f\) option causes the Naming Service to use a memory-mapped file as its storage mechanism. This file is a copy of the internal memory used to store the naming graph. Thus, the \(-s\) option will have some effect on the size of the single file created.

The \(-u\) and \(-r\) options both use the same flat-file storage scheme. In this scheme, a separate file is created for each naming context in the naming graph. All of the data for a single naming context is stored in a single file, including both object references of bound objects and references to other naming contexts. These files are plain text and of variable sizes. No special provisions have been made for internationalization at this time.

The argument provided with both the \(-u\) and \(-r\) options is the name of the directory in which the persistence files are to be created. The directory must already exist. Two text files are initially created in this directory. The first file, called `NameService`, stores the name bindings in the root naming context. The second file, called `NameService_global`, stores a count of children naming contexts created from the root naming context. The initial value of this counter is platform dependent. For every new naming context created, the counter in `NameService_global` is incremented by one and a new text file called `NameService_num` is created. The value of `num` is the integer obtained by decrementing by one the counter stored in `NameService_global`.

The main difference between the \(-u\) and \(-r\) persistence options is that the \(-r\) option uses file locking while the \(-u\) option does not. The use of file locking allows the files to be accessed safely by more than one Naming Service server running concurrently, however it has a small impact on performance. The flat-file persistence implementation (\(-u\)) does not acquire file locks.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-z) time</td>
<td>Specify a request/reply round trip timeout value that the Naming Service will use when invoking an operation on a federated naming context. On timeout, a <code>CosNaming::NamingContext::CannotProceed</code> exception is raised. The value of <code>time</code> is expressed in seconds.</td>
<td>A timeout policy is not used and an exception is not raised.</td>
</tr>
</tbody>
</table>
The same base implementation is used for the flat-file (-u) and redundant (-r) persistence options. This implementation makes use of a PortableServer::ServantActivator that activates servants to read the naming context files only when they are used. This implementation was required for the redundant case because a naming context may have been created in one Naming Service server and referenced from another. To make this work, naming context references are stored as textual names rather than stringified IORs. Each textual name corresponds to the name of the file that stores the naming context’s data. The object reference for a naming context is created dynamically within each Naming Service server when the naming context is accessed.

The flat-file storage format consists of a persistent_header followed by zero or more persistent_records as shown in Figure 24-3. The persistent_header contains information about the context, whereas each persistent_record represents a name-to-reference binding. Table 24-2 describes the fields that make up a persistent_header and a persistent_record.
### Table 24-2 Flat-File Persistence Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE_FIELD</td>
<td>An integer that represents the number of PERSISTENT_RECORDs in a naming context.</td>
</tr>
<tr>
<td>DESTROYED_FIELD</td>
<td>A flag set to 1 when the context is destroyed and 0 otherwise.</td>
</tr>
<tr>
<td>TYPE_FIELD</td>
<td>An integer that specifies what kind of reference is contained in the REF_FIELD. A value of 0 specifies that the field contains a reference to a naming context. A value of 1 specifies that the field contains a reference to a regular object.</td>
</tr>
<tr>
<td>ID_LENGTH</td>
<td>An integer that represents the length of the ID_FIELD.</td>
</tr>
</tbody>
</table>

**Figure 24-3 Persistent Flat-File Format**
The redundant naming service is only fully functional on an HP Tru64 UNIX cluster, which was the target platform for its implementation. In a Tru64 cluster, multiple nodes share a single IP address; additional facilities within the cluster route requests to multiple redundant servers without outside intervention. Since naming context object references contain an IP address, the use of a single IP address for all the nodes of the cluster allows the naming context object references to be valid no matter which machine actually processes the request.

In the redundant naming service, the disk files become the single place where authoritative information can be found. This requires that each node that is running an instance of the Naming Service server have access to the same files and that a locking mechanism must be available to protect files from simultaneous access. The Tru64 cluster provides these facilities.

Though the redundant Naming Service implementation was targeted for the Tru64 cluster environment, it can also be used on non-clustered platforms as long as certain restrictions are carefully observed:

1. There must be a shared location in which to store the flat files, and a file locking mechanism must be available. The NFS with a locking daemon satisfies this requirement on UNIX and UNIX-like systems. The SMB and built-in locking satisfy this requirement for Windows platforms.

2. The client must explicitly select one of the redundant Naming Service servers to use. Thus, a given client will generally use a single Naming Service server unless it fails. Then, the client must fail-over to a different server. The fail-over mechanism is not inherent to the redundant Naming Service and must be implemented within the client.

Table 24-2 Flat-File Persistence Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_FIELD</td>
<td>A string that represents the id field of a CosNaming::NameComponent in a binding.</td>
</tr>
<tr>
<td>KIND_LENGTH</td>
<td>An integer that represents the length of the KIND_FIELD.</td>
</tr>
<tr>
<td>KIND_FIELD</td>
<td>A string that represents the kind field of a CosNaming::NameComponent in a binding.</td>
</tr>
<tr>
<td>REF_LENGTH</td>
<td>An integer that represents the length of the REF_FIELD.</td>
</tr>
<tr>
<td>REF_FIELD</td>
<td>A string that represents the object reference portion of a binding. If the binding is to a naming context, this field will be the name of file that stores that naming context’s data. If the binding is to a regular object, this field will be a stringified IOR.</td>
</tr>
</tbody>
</table>
3. If a client switches to a different Naming Service server, all Naming Context object references it is holding will no longer be valid and the client must start over with the root Naming Context of the new Naming Service. Clients must be careful to use Naming Context object references only with the Naming Service server from which they were obtained.

4. `CosNaming::BindingIterator` object references, such as those returned from the `CosNaming::NamingContext::list()` operation, are not usable with the redundant Naming Service implementation. The `list()` operation can still be used, but only bindings in the returned `CosNaming::BindingList` are usable.

The performance of the Naming Service is impacted by any use of persistence. Memory-mapped persistence (the `-f` option) has the smallest impact; then non-redundant flat-file persistence (the `-u` option); redundant flat-file persistence (the `-r` option) has the largest impact on performance.

### 24.8.2 Example using the Naming Service Persistence Options

In this section, we present a simple example of using the Naming Service persistence options. We revisit the Naming Service example presented in 24.3. For this example, we define a new environment variable called `NAMING_DIRECTORY` as follows:

```bash
NAMING_DIRECTORY=$TAO_ROOT/DevGuideExamples/NamingService/Messenger
export NAMING_DIRECTORY
```

Start the Naming Service as in 24.3.3.2. Specify either the `-u` or `-r` persistence option, for example:

```bash
$ Naming_Service -ORBListenEndpoints iiop://tango:2809 -u $NAMING_DIRECTORY
```

Next, run the `MessengerServer` and `MessengerClient` as in 24.3.3.2. You should find that the following files are created in the directory specified by `$NAMING_DIRECTORY`:

```
NameService  NameService_0  NameService_global
```

The contents the `NameService` file below indicate that the root context has not been destroyed and that it contains a single binding. The `id` field of the
Naming Service

binding is “example” and the binding is for a naming context, the contents of which are stored in a file named NameService_0.

Similarly, the contents of the NameService_0 file below indicate that this naming context has not been destroyed and contains a single name binding. The id field of the binding is "Messenger" and the binding is for a regular object. The stringified object reference is listed as well (it may be different on your system).

Since we started the Naming Service using the -ORBListenEndpoints option described in 19.8.10, if we later need to restart the Naming Service server, we must restart it on the same endpoint(s) as before.
25.1 Introduction

The OMG Event Service version 1.2 specification (OMG Document formal/04-10-02) defines a service for decoupling the suppliers of events from consumers of those events. This decoupled approach provides a much more appropriate communication model for many applications than the typical request/reply semantics of CORBA object operation invocations. The Event Service defines basic interfaces for suppliers and consumers of events and defines the concept of an event channel to provide for decoupling consumers from suppliers and propagating events.

This chapter discusses TAO's support for the Event Service as well as how to use, extend, and embed the Event Service in your applications. TAO also defines two extensions to the Event Service. The Real-Time Event Service (see Chapter 26) extends the OMG’s specification for applications with stringent Quality of Service requirements. The OMG Notification Service (see Chapter 27) is a more recent specification than the Event Service, and extends the basic Event Service functionality with features such as event filtering, structured event types, and Quality of Service properties.
25.2 Overview of the Event Service

The Event Service is based on a publish and subscribe paradigm, where suppliers publish events and consumers receive events for which they have subscribed. An event channel provides a mechanism that decouples suppliers from consumers. A supplier publishes events via the event channel, and a consumer subscribes to them through the event channel. Suppliers are not directly aware of the presence of consumers (nor other suppliers). Similarly, consumers are not directly aware of the presence of suppliers (nor other consumers).

Figure 25-1 shows the relationships among suppliers, consumers, and an event channel, and illustrates typical usage of an event channel to distribute events.

![Figure 25-1 Typical Event Channel Usage]

The Event Service specification supports push and pull style event distribution models. In the push model, suppliers push events to the event channel, and the event channel pushes them to all subscribed consumers. In the pull model, consumers pull events from the event channel, and the event channel pulls events from suppliers. The Event Service specification makes it possible to mix the push and pull models in different combinations, even within the same event channel.

The Event Service specification defines two different varieties of event channels, typed and untyped. Untyped event channels use a CORBA::Any type to represent events. This allows suppliers to supply arbitrary event types. Typed event channels use application-defined IDL to define the event communications and constrain the data types passed.
25.3 TAO’s Event Channel Implementation

See Chapter 20 of *Advanced CORBA Programming with C++* for an in-depth discussion of the Event Service and the different distribution models.

25.3 TAO’s Event Channel Implementation

TAO provides support for untyped event channels via the `CosEventChannelAdmin::EventChannel` interface. This implementation supports untyped events using `CORBA::Any` parameters with both the push and pull models of event delivery. See 25.4.1 for an example that uses an untyped event channel.

TAO also provides partial support for typed event channels via the `CosTypedEventChannelAdmin::TypedEventChannel` interface. This implementation supports typed event delivery using a push model. The pull model is not currently supported for typed event channels. See 25.4.4 for an example that uses a typed event channel.

TAO does not support the Lightweight Event Service described in Chapter 3 of the Event Service specification.

The `CosEvent_Service` server can be used to start both typed and untyped event channels (see 25.5 for details). Developers can directly create or derive their own untyped event channels using the `TAO_CEC_EventChannel` servant class (see 25.4.2).

25.4 How to Use the Event Service

The examples in this chapter restrict themselves to the basic capabilities and interfaces of the Event Service. TAO-specific steps are noted as such. Because these examples use the Naming Service to locate the event channel, ensure that it is running as described in Chapter 24.

25.4.1 A Basic Example

This example shows how to create an event channel, connect suppliers and consumers to it, supply events to it, and consume events from it. It uses push suppliers and consumers. Full source code for this example is in the TAO 1.4a source code distribution in the directory `$TAO_ROOT/DevGuideExamples/EventServices/OMG_Basic`. 
25.4.1.1 Starting the CosEvent_Service Server

The CosEvent_Service server that is delivered with TAO can be used with this example. This server creates a single event channel object and binds it in the root naming context of the Naming Service. By default, the event channel is bound to the name “CosEventService.” The options that can be passed to this server are described in 25.5 and 25.6. A typical invocation of the server is:

```
$TAO_ROOT/orbsvcs/CosEvent_Service/CosEvent_Service
```

This assumes that we are using either multicast discovery or the NameServiceIOR environment variable to locate the Naming Service.

25.4.1.2 Creating and Initializing a Supplier and Pushing Events

To implement a supplier, insert the following #include directives in your source code (in EchoEventSupplierMain.cpp):

```c++
#include <orbsvcs/CosEventCommC.h>
#include <orbsvcs/CosEventChannelAdminC.h>
#include <orbsvcs/CosNamingC.h>
#include <iostream>
```

The supplier must first initialize the ORB, then connect to the event channel. Since the event channel object is now bound in the Naming Service, we need to get the root naming context and use the `resolve_str()` operation to obtain a proxy for the event channel.

```c++
int main (int argc, char* argv[]) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Find the Naming Service.
        CORBA::Object_var obj = orb->resolve_initial_references("NameService");
        CosNaming::NamingContextExt_var root_context =
            CosNaming::NamingContextExt::_narrow(obj.in());

        // Find the EventChannel.
        obj = root_context->resolve_str("CosEventService");

        // Narrow the object reference to an EventChannel reference.
        CosEventChannelAdmin::EventChannel_var echoEC =
```
25.4 How to Use the Event Service

CosEventChannelAdmin::EventChannel::_narrow(obj.in());
if (CORBA::is_nil(echoEC.in())) {
    std::cerr << "Could not resolve EchoEventChannel." << std::endl;
    return 1;
}

Once the event channel is located, it is used to obtain a proxy to a push consumer. The proxy push consumer is then used to connect to the event channel.

// Get a SupplierAdmin object from the EventChannel.
CosEventChannelAdmin::SupplierAdmin_var supplierAdmin =
    echoEC->for_suppliers();

// Get a ProxyPushConsumer from the SupplierAdmin.
CosEventChannelAdmin::ProxyPushConsumer_var consumer =
    supplierAdmin->obtain_push_consumer();

// Connect to the ProxyPushConsumer as a PushSupplier
// (passing a nil PushSupplier object reference to it because
// we don't care to be notified about disconnects).
consumer->connect_push_supplier(CosEventComm::PushSupplier::_nil());

The connect_push_supplier() operation takes a reference to a push supplier as a parameter. The only operation defined on the push supplier interface is disconnect_push_supplier(). It is called when the supplier is disconnected from the event channel. Passing a null object reference, as shown above, means the event channel does not need to notify this supplier upon disconnection. For our simple example, we do not need to implement the PushSupplier interface. We show how this could be done in 25.4.1.3.

We are now ready to create and publish events using the consumer proxy’s push() operation.

// Create an event (just a string in this case).
CORBA::String_var eventData = CORBA::string_dup("Hello, world.");

// Insert the event data into an any.
CORBA::Any any;
any <<= eventData;

// Now push the event to the consumer
consumer->push(any);
}
catch { ... }
return 1;
Our example passes a simple string as the event data. However, since the event is passed as a CORBA::Any, virtually any IDL type (including user-defined types) can be used as event types.

### 25.4.1.3 Implementing the Push Supplier Interface

In our simple example, we have chosen to make the supplier a pure client of the CosEvent service. However, we could also implement the CosEventComm::PushSupplier interface to allow our supplier to receive disconnect_push_supplier() callbacks from the CosEvent service. In practice, suppliers are rarely pure clients; they are often middle-tier processes that receive data directly from one or more “raw” data sources and publish the data as events for further processing or display by downstream consumers.

Here is an example PushSupplier implementation class definition:

```cpp
#include <orbsvcs/CosEventCommS.h> // for POA_CosEventComm::PushSupplier

class EchoEventSupplier_i : public virtual POA_CosEventComm::PushSupplier
{
  public:
    // Constructor
    EchoEventSupplier_i(CORBA::ORB_ptr orb);

    // Override operations from PushSupplier interface.
    virtual void disconnect_push_supplier()
    throw(CORBA::SystemException);

  private:
    CORBA::ORB_var orb_;  
};
```

Here are the implementations of the constructor and the disconnect_push_supplier() operation:

```cpp
// Constructor duplicates the ORB reference.
EchoEventSupplier_i::EchoEventSupplier_i(CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb))
{
}

// Override the disconnect_push_supplier() operation.
void EchoEventSupplier_i::disconnect_push_supplier()
  throw(CORBA::SystemException)
```
25.4 How to Use the Event Service

25.4.1.4 Implementing the Push Consumer Interface

To create a consumer, the CosEventComm::PushConsumer interface must be implemented. The PushConsumer interface defines two operations, push() and disconnect_push_consumer(), that you must implement. Here is an example PushConsumer implementation (from EchoEventConsumer_i.h) class definition:

```cpp
#include <orbsvcs/CosEventCommS.h>    // for POA_CosEventComm::PushConsumer

class EchoEventConsumer_i : public virtual POA_CosEventComm::PushConsumer
{
public:
    // Constructor
    EchoEventConsumer_i(CORBA::ORB_ptr orb);

    // Override operations from PushConsumer interface.
    virtual void push(
        const CORBA::Any& data)
        throw(CORBA::SystemException);

    virtual void disconnect_push_consumer()
        throw(CORBA::SystemException);

private:
    CORBA::ORB_var orb_;
};
```

Here are the implementations of the constructor, the push() operation and the disconnect_push_consumer() operation (from EchoEventConsumer_i.cpp):

```cpp
// Constructor duplicates the ORB reference.
EchoEventConsumer_i::EchoEventConsumer_i(CORBA::ORB_ptr orb)
    : orb_(CORBA::ORB::_duplicate(orb))
{}
// Override the push() operation.
void EchoEventConsumer_i::push(
    const CORBA::Any& data)
throw(CORBA::SystemException)
{
    // Extract event data from the Any.
    const char* eventData;
    if (data >>= eventData) {
        std::cout << "EchoEventConsumer_i::push(): Received event: "
                   << eventData << std::endl;
    }
}

// Override the disconnect_push_consumer() operation.
void EchoEventConsumer_i::disconnect_push_consumer()
throw(CORBA::SystemException)
{
    // Deactivate this object.
    CORBA::Object_var obj = orb_->resolve_initial_references("POACurrent");
    PortableServer::Current_var current =
        PortableServer::Current::_narrow(obj.in());
    PortableServer::POA_var poa = current->get_POA();
    PortableServer::ObjectId_var object_id = current->get_object_id();
    poa->deactivate_object(object_id.in());
}

25.4.1.5 Creating the Consumer and Connecting to the Channel
To receive events, the application must now create a consumer object, connect
to the event channel, and enter the event loop. Locating the event channel is
exactly the same as on the supplier side (see
EchoEventSupplierMain.cpp):

int main (int argc, char* argv[])
{
    try
    {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Find the Naming Service.
        CORBA::Object_var obj = orb->resolve_initial_references("NameService");
        CosNaming::NamingContextExt_var root_context =
            CosNaming::NamingContextExt::_narrow(obj.in()));

        // Find the EchoEventChannel.
        obj = root_context->resolve_str("CosEventService");

        // Narrow the object reference to an EventChannel reference.
CosEventChannelAdmin::EventChannel_var echoEC =
    CosEventChannelAdmin::EventChannel::_narrow(obj.in());
if (CORBA::is_nil(echoEC.in())) {
    std::cerr << "Could not narrow EchoEventChannel." << std::endl;
    return 1;
}
std::cout << "Found the EchoEventChannel." << std::endl;

Next, we create a consumer to receive the events:

    // Instantiate an EchoEventConsumer_i servant.
    EchoEventConsumer_i servant(orb.in());

    // Register it with the RootPOA.
    obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
    PortableServer::ObjectId_var oid = poa->activate_object(&servant);
    CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
    CosEventComm::PushConsumer_var consumer =
        CosEventComm::PushConsumer::_narrow(consumer_obj.in());

Now we obtain a ProxyPushSupplier and connect the consumer to the event channel:

    // Get a ConsumerAdmin object from the EventChannel.
    CosEventChannelAdmin::ConsumerAdmin_var consumerAdmin =
        echoEC->for_consumers();

    // Get a ProxyPushSupplier from the ConsumerAdmin.
    CosEventChannelAdmin::ProxyPushSupplier_var supplier =
        consumerAdmin->obtain_push_supplier();

    // Connect to the ProxyPushSupplier, passing your PushConsumer object
    // reference to it.
    supplier->connect_push_consumer(consumer.in());

After activating the POA and starting the event loop, the consumer is now ready to receive events.

    // Activate the POA via its POAManager
    PortableServer::POAManager_var poa_manager = poa->the_POAManager();
    poa_manager->activate();
    std::cout << "Ready to receive events..." << std::endl;

    // Enter the ORB event loop.
    orb->run();
    orb->destroy();
The `push()` operation of the consumer is now invoked each time a supplier pushes an event onto this event channel.

### 25.4.2 Creating and Configuring Event Channel Servants

The `CosEvent_Service` server provides a simple and convenient way to create event channels and a means for suppliers and consumers to find and connect to these channels (via the Naming Service). This approach is sufficient for simple examples like the one above. However, you may want to create and manage your own event channel servants in application processes to:

- Implement multiple event channels in one process.
- Collocate the event channel with a supplier or consumer.
- Provide a different mechanism (other than simple names in the root naming context) for locating event channels.
- Easily and efficiently control the creation and destruction of event channels.
- Federate your event channel with other event channels.
- Customize the behavior of the event channel.

To create and manage an event channel servant that is based on the implementation, use the `TAO_CEC_EventChannel` class. You can either directly instantiate one of these objects or derive your own subclass and specialize its behavior. The following example shows a supplier process that uses the `TAO_CEC_EventChannel` class to create its own local event channel.

#### 25.4.2.1 Supplier/EC Collocation Example

Most of the code for this example is the same as in the previous example. Here we show only those sections of code that differ. Full source code for this example is in the TAO 1.4a source code distribution in the directory

```cpp
} catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
}

The push() operation of the consumer is now invoked each time a supplier pushes an event onto this event channel.

### 25.4.2 Creating and Configuring Event Channel Servants

The `CosEvent_Service` server provides a simple and convenient way to create event channels and a means for suppliers and consumers to find and connect to these channels (via the Naming Service). This approach is sufficient for simple examples like the one above. However, you may want to create and manage your own event channel servants in application processes to:

- Implement multiple event channels in one process.
- Collocate the event channel with a supplier or consumer.
- Provide a different mechanism (other than simple names in the root naming context) for locating event channels.
- Easily and efficiently control the creation and destruction of event channels.
- Federate your event channel with other event channels.
- Customize the behavior of the event channel.

To create and manage an event channel servant that is based on the implementation, use the `TAO_CEC_EventChannel` class. You can either directly instantiate one of these objects or derive your own subclass and specialize its behavior. The following example shows a supplier process that uses the `TAO_CEC_EventChannel` class to create its own local event channel.

#### 25.4.2.1 Supplier/EC Collocation Example

Most of the code for this example is the same as in the previous example. Here we show only those sections of code that differ. Full source code for this example is in the TAO 1.4a source code distribution in the directory

```cpp`
Any process containing event channel servants must make the following call before calling CORBA::ORB_init():

// Initialize the CEC Factory so we can customize the CEC
TAO_CEC_Default_Factory::init_svcs();

This code initializes the factory that the servant uses to configure itself (see 25.6 for details). Because this configuration happens via the service configurator, the factory must be initialized before the service configurator is initialized (which occurs during the ORB_init() call).

The remaining changes replace the code in the supplier that resolves the event channel from the Naming Service and narrows its object reference.

// Get the RootPOA
CORBA::Object_var poa_object = orb->resolve_initial_references("RootPOA");

PortableServer::POA_var poa =
    PortableServer::POA::narrow (poa_object.in());
PortableServer::POAManager_var poa_manager = poa->the_POAManager();
poa_manager->activate();

// Create and activate the event channel servant
TAO_CEC_EventChannel_Attributes attr(poa.in(), poa.in());
TAO_CEC_EventChannel* ec = new TAO_CEC_EventChannel(attr);
ec->activate();
PortableServer::ObjectId_var oid = poa->activate_object(ec);
CORBA::Object_var ec_obj = poa->id_to_reference(oid.in());
CosEventChannelAdmin::EventChannel_var echoEC =
    CosEventChannelAdmin::EventChannel::_narrow(ec_obj.in());

// Bind the EventChannel in the Naming Service.
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("CosEventService");
root_context->rebind(name, echoEC.in());

The first block of code locates the root POA and activates it via the POA manager. This is required because the supplier process must now become a CORBA server.

Next, the event channel servant is constructed, initialized, and activated. The TAO_CEC_EventChannel_Attributes class is used to initialize the event
channel. The code in this example simply tells the event channel to use the root POA when activating new CORBA objects. In 25.4.2.2 we describe in more detail the usage of the TAO_CEC_EventChannel_Attributes class.

Lastly, the event channel is bound to a name in the Naming Service so that consumers can locate it.

Overall, this example executes in the same way as the previous one, with the exception that it is slightly more efficient because of the collocation of the event channel and supplier. An additional advantage is that an alternative mechanism for advertising the existence and location of the event channel can be used (e.g., writing the IOR as a string to a file, or advertising it via the Trading Service).

25.4.2.2 Setting Attributes of the Event Channel

The event channel has a number of attributes that are set via the TAO_CEC_EventChannel_Attributes object that is passed to the constructor of the event channel. Table 25-1 provides a summary of the attributes that can be set:

Table 25-1 Event Channel Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>supplier_poa</td>
<td>PortableServer::POA_ptr</td>
<td>None</td>
<td>POA used by supplier admin and supplier proxies. This is typically the same POA the EC uses.</td>
</tr>
<tr>
<td>consumer_poa</td>
<td>PortableServer::POA_ptr</td>
<td>None</td>
<td>POA used by consumer admin and consumer proxies. This is typically the same POA the EC uses.</td>
</tr>
<tr>
<td>consumer_reconnect</td>
<td>int</td>
<td>0</td>
<td>Enables consumer reconnections when non-zero.</td>
</tr>
<tr>
<td>supplier_reconnect</td>
<td>int</td>
<td>0</td>
<td>Enables supplier reconnections when non-zero.</td>
</tr>
<tr>
<td>disconnect_callbacks</td>
<td>int</td>
<td>0</td>
<td>If not zero, the event channel sends disconnect callbacks when a disconnect operation is called on a proxy.</td>
</tr>
</tbody>
</table>

In the previous example, the only attributes set are the supplier and consumer POAs. These are set via the EventChannel_Attributes constructor as follows:
25.4 How to Use the Event Service

```cpp
TAO_CEC_EventChannel_Attributes attributes (poa.in (), // Supplier POA
  poa.in ()); // Consumer POA

TAO_CEC_EventChannel* ec = new TAO_CEC_EventChannel(attributes);
```

The event channel uses the supplier POA to activate `SupplierAdmin`, `ProxyPushSupplier`, and `ProxyPullSupplier` servants, and the consumer POA to activate `ConsumerAdmin`, `ProxyPushConsumer`, and `ProxyPullConsumer` servants. In our case, we pass the same POA object reference for use as both the supplier POA and the consumer POA.

All other attributes of the event channel are set using public data members of the `TAO_CEC_EventChannel_Attributes` class. For example:

```cpp
TAO_CEC_EventChannel_Attributes attributes (poa.in (), poa.in ());
attributes.disconnect_callbacks = 1;
TAO_CEC_EventChannel* ec_impl = new TAO_CEC_EventChannel(attributes);
```

The `disconnect_callbacks` attribute controls whether the consumer and supplier disconnect callbacks are called when the corresponding disconnect operation is called on the proxy object. For example, if this attribute is set to true, then when a consumer calls `disconnect_push_supplier()` on its proxy, the event channel invokes `disconnect_push_consumer()` on the consumer. Similar behaviors exist for pull suppliers as well as both types of consumers. It is a matter of debate as to whether the Event Service specification requires these callbacks to be made. These callbacks are always called when the event channel initiates the disconnection.

The `supplier_reconnect` and `consumer_reconnect` attributes allow suppliers and consumers to call the operations `connect_push_supplier()` and `connect_push_consumer()` multiple times without disconnecting. This allows you to more efficiently replace the supplier or consumer object passed to the event channel.

The default values for these attributes are defined as preprocessor macros in the `$TAO_ROOT/orbsvcs/orbsvcs/CosEvent/CEC_Defaults.h`. You can use your own project-specific defaults by setting these macros in your `config.h` file and recompiling the event service.
25.4.3 **Pull Model Support**

The Event Service Specification gives a great deal of latitude in the terms of how an implementation supports the pull model. This section describes the details of TAO’s implementation of the pull model.

When pull suppliers are connected, the event channel periodically attempts to pull events from each pull supplier. The default period between attempted pulls is 5 seconds. This value can be modified via a service configurator option, see 25.6.1.5 for details.

Pushed and pulled events are immediately delivered to all push consumers and are queued for delivery to pull consumers. The event channel maintains a separate event queue for each pull consumer. When pull consumers attempt to pull events, the oldest event is removed from its queue and returned. If the queue is empty, then the call blocks until an event is available.

---

**Note**  
TAO’s implementation of the CosEvent service’s pull model works reliably only when the service is using the thread-per-connection concurrency model. See 21.3.5 for more information on how to configure this behavior.

25.4.4 **Typed Event Channel Example**

This example shows how a typed event channel can be used with push consumers and suppliers. We utilize a modified version of the Messenger interface that is used throughout this book. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/EventServices/OMG_TypedEC.

25.4.4.1 **The Messenger Interface**

In order to use a typed event channel, your application must define an IDL interface that constrains the event data that is passed through the EC. This is a normal IDL interface but with certain restrictions on the operations it contains. The operations contained in the interface must not have return values and can only contain in parameters. These are effectively the same restrictions as those placed on oneway operations (although operations on the typed EC interface can be oneway or synchronous).

Here is the Messenger interface with the modifications necessary to allow its usage with a typed event channel.
### 25.4.4.2 Creating the Typed Event Channel

TAO’s typed event channel implementation utilizes the Interface Repository to allow it take a single invocation on our application interface from a particular supplier and propagate it to multiple consumers. Therefore, we need to start the Interface Repository server and load the interface description of our Messenger interface into it before creating and using the typed event channel. We use the following commands to start and populate the Interface Repository:

```bash
export InterfaceRepositoryIOR=file://ifr.ior
$TAO_ROOT/orbsvcs/IFR_Service/IFR_Service -o ifr.ior &
$ACE_ROOT/bin/tao_ifr Messenger.idl
```

For more information about TAO’s Interface Repository see Chapter 28. Now, using the CosEvent_Service server, we can create the typed event channel.

```bash
$TAO_ROOT/orbsvcs/CosEvent_Service/CosEvent_Service -t &
```

Just as in the previous examples, this stores the event channel object reference in the naming service with the only difference being the -t option which means that the event channel now implements the `CosTypedEventChannelAdmin::TypedEventChannel` interface.

### 25.4.4.3 Implementing the Typed Supplier

Most of the supplier code remains very similar to the untyped supplier we saw previously. The main differences are the use of the corresponding “typed” interfaces in this example and the details about how typed events are published. First, we substitute `#include` directives for the typed event channel IDL. We also include the `Typecode.h` header, so we can later use the `TypeCode` interface to get the repository ID of the Messenger interface.

```c
#include <orbsvcs/CosTypedEventCommC.h>
#include <orbsvcs/CosTypedEventChannelAdminC.h>
```
Event Service

```c++
#include <tao/Typecode.h>

The first differences in the body of the program are where the event channel is
narrowed as it is now a TypedEventChannel.

// Find the EventChannel.
obj = root_context->resolve_str("CosEventService");

// Downcast the object reference to a TypedEventChannel reference.
CosTypedEventChannelAdmin::TypedEventChannel_var ec =
    CosTypedEventChannelAdmin::TypedEventChannel::_narrow(obj.in ());

Connecting to the event channel goes through the same steps as the untyped
example: get the supplier admin, obtain a proxy consumer, and connect to the
proxy consumer.

// Get a SupplierAdmin object from the EventChannel.
CosTypedEventChannelAdmin::TypedSupplierAdmin_var supplierAdmin =
    ec->for_suppliers();

// Get a ProxyPushConsumer from the SupplierAdmin.
CosTypedEventChannelAdmin::TypedProxyPushConsumer_var consumer =
    supplierAdmin->obtain_typed_push_consumer(::tc_Messenger->id());

// Connect to the ProxyPushConsumer as a PushSupplier
// (passing a nil PushSupplier object reference to it because
// we don't care to be notified about disconnects).
consumer->connect_push_supplier(CosEventComm::PushSupplier::_nil());

One key difference above is that the obtain_typed_push_consumer() operation which takes a string parameter that specifies the IDL interface type
by passing its repository ID. We obtain the Messenger interface’s repository
ID by using the TypeCode object’s id() operation.

In order to publish typed events, we now need to obtain an object reference
that implement’s the Messenger interface. The TypedProxyPushConsumer
interface supports a get_typed_consumer() operation that returns an
object reference that implements the type associated with that proxy.

// Obtain the interface from the event channel
CORBA::Object_var messenger_obj = consumer->get_typed_consumer();

// Narrow the interface
Messenger_var messenger = Messenger::_narrow(messenger_obj.in () );
```
We can now publish events to any connected consumers by simply invoking the `send_message()` operation on our Messenger object reference. The typed event channel implementation is responsible for multiplexing this operation so that all eligible consumers that receive the same request.

```c++
// Send one event per second. (approx)
while (1) {
    messenger->send_message("King Lizard",
                             "Proclamations",
                             "Hello, world");

    ACE_Time_Value event_delay(0, 1000 * EVENT_DELAY_MS);
    orb->run(event_delay);
}
```

**Note** Because the `TypedProxyPushConsumer` interface is derived from the `ProxyPushConsumer` interface it also implements the `push()` operation. According to the Event Service specification, suppliers to typed event channels can also publish untyped events by utilizing this operation. TAO’s typed event channel implementation does not support this feature and any attempt to call `push()` on a typed proxy push consumer results in a `NO_IMPLEMENT` exception.

### 25.4.4.4 Implementing the Typed Consumer

Implementing the typed consumer is slightly more complicated than the untyped consumer. We still need to implement a consumer servant and connect it to the event channel, but now we also have to implement our application-specific interface (Messenger in our case) and associate it with our consumer.

First, let’s look at our messenger servant.

```c++
class Messenger_i : public virtual POA_Messenger {
public:
    Messenger_i (CORBA::ORB_ptr orb, int event_limit);
    virtual ~Messenger_i () {

        virtual void send_message (const char * user_name,
                                   const char * subject,
                                   const char * message)
        throw (CORBA::SystemException);

    private:
```
The constructor takes two arguments, an ORB reference and an event limit. Both values are stored in data members for use by `send_message()`, which prints each message’s data and shuts down the ORB when the event limit is reached.

```cpp
void Messenger_i::send_message (const char * user_name,
                                    const char * subject,
                                    const char * message)
                                    throw (CORBA::SystemException)
{
    std::cout << "Message from: " << user_name << std::endl;
    std::cout << "Subject: " << subject << std::endl;
    std::cout << "Message: " << message << std::endl;

    if (--event_limit_ <= 0) { 
        orb_->shutdown(0);
    }
}
```

Next, let’s look at our typed consumer servant which we’ll connect to the event channel and use to hold our Messenger object reference.

```cpp
#include <orbsvcs/CosTypedEventCommS.h>

class Consumer_i : public virtual POA_CosTypedEventComm::TypedPushConsumer {
public:
    Consumer_i(CORBA::ORB_ptr orb,
               CORBA::Object_ptr obj);

    // Override operations from TypedPushConsumer interface.
    virtual CORBA::Object_ptr get_typed_consumer ()
        throw (CORBA::SystemException);

    virtual void push(const CORBA::Any & data)
        throw(CORBA::SystemException);

    virtual void disconnect_push_consumer()
        throw(CORBA::SystemException);

private:
    CORBA::ORB_var orb_;
The `disconnect_push_consumer()` member function is implemented as before for untyped consumers. The `push()` member function is used by the event channel to deliver any untyped events published to that channel. If we do not wish to receive untyped events we may simply throw a `NO_IMPLEMENT` exception.

```cpp
void Consumer_i::push(const CORBA::Any & data) throw(CORBA::SystemException) {
    throw CORBA::NO_IMPLEMENT();
}
```

**Note** Because TAO does not support publication of untyped events via typed event channels, this operation should never be called when using TAO's typed event channel implementation.

The `get_typed_consumer()` member function should return an object reference to a CORBA object that implements the specific interface we are using for this consumer. We pass the object reference as a constructor parameter, store it in a data member, and return it from `get_typed_consumer()`. By using `CORBA::Object` as the object reference data type, we allow this class to be potentially used with other interface types (besides `Messenger`).

```cpp
Consumer_i::Consumer_i(CORBA::ORB_ptr orb, CORBA::Object_ptr obj)
    : orb_(CORBA::ORB::_duplicate(orb)), object_(CORBA::Object::_duplicate(obj))
{
}
CORBA::Object_ptr
Consumer_i::get_typed_consumer ()
    throw (CORBA::SystemException)
{
    return CORBA::Object::_duplicate(object_.in());
}
```

### 25.4.4.5 Connecting the Typed Consumer

Connecting our typed consumer to the event channel is very similar to the untyped example with the important differences being the use of “typed”
interfaces and some details of the consumer object construction. First, we see some different headers we’ll need:

```
#include <orbsvcs/CosTypedEventCommC.h>
#include <orbsvcs/CosTypedEventChannelAdminC.h>
#include <tao/Typecode.h>
```

After we get the event channel reference from the naming service, we can narrow it to a `TypedEventChannel`, get the consumer admin reference, and obtain a typed proxy supplier.

```
obj = root_context->resolve_str("CosEventService");

// Downcast the object reference to a TypedEventChannel reference.
CosTypedEventChannelAdmin::TypedEventChannel_var ec =
    CosTypedEventChannelAdmin::TypedEventChannel::_narrow(obj.in());

// Get a ConsumerAdmin object from the EventChannel.
CosTypedEventChannelAdmin::TypedConsumerAdmin_var consumerAdmin =
    ec->for_consumers();

// Get a ProxyPushSupplier from the ConsumerAdmin.
CosEventChannelAdmin::ProxyPushSupplier_var supplier =
    consumerAdmin->obtain_typed_push_supplier(::tc_Messenger->id());
```

We used the repository ID of the Messenger interface when obtaining the typed proxy supplier which associates that interface type with the consumer when we connect to that proxy. Now we are ready to create our consumer and connect it to the event channel.

```
// Get the RootPOA.
// Activate the POA manager here before we connect our consumer.
obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
PortableServer::POAManager_var poa_manager = poa->the_POAManager();
poa_manager->activate();

// Create our Messenger_i servant and activate the CORBA object
Messenger_i servant(orb.in(), EVENTS_TILL_SHUTDOWN);
PortableServer::ObjectId_var oid = poa->activate_object(&servant);
CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());

// Create our Consumer servant and pass it the Messenger
// object reference. Activate the consumer CORBA object.
Consumer_i consumer_servant(orb.in(), messenger_obj.in());
PortableServer::ObjectId_var cons_oid =
```
25.5 CosEvent_Service Command Line Options

The CosEvent_Service server supplies the capability to start a single event channel in its own process. It can bind the created event channel to a supplied name in the root naming context of the Naming Service. The Naming Service must be running to use this server (unless the -x option is used). Table 25-2 describes the available command line options.

Table 25-2 CosEvent_Service Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-n COS_EC_name</td>
<td>Specifies the name with which to bind the event channel (in the root naming context of the Naming Service).</td>
<td>CosEventService</td>
</tr>
<tr>
<td>-o filename</td>
<td>Specify the name of the output file for writing the Event Channel’s IOR as a string.</td>
<td>Do not write the IOR to a file.</td>
</tr>
<tr>
<td>-r</td>
<td>Use the rebind() operation to bind the event channel in the Naming Service. If the name is already bound, and this flag is not passed, then the process exits with an Already Bound exception.</td>
<td>The bind() operation is used.</td>
</tr>
</tbody>
</table>
Table 25-2 CosEvent_Service Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>Do not use the Naming Service. This simply creates an event channel.</td>
<td>Bind the EC in the Naming Service.</td>
</tr>
<tr>
<td>-t</td>
<td>Create a typed event channel</td>
<td>Create an untyped event channel</td>
</tr>
<tr>
<td>-d</td>
<td>Destroy flag for typed event channels. Determines whether shutdown of a typed EC shuts down the ORB.</td>
<td>Don’t shutdown the ORB for typed ECs.</td>
</tr>
</tbody>
</table>

When -t is specified, the event channel object implements the `CosTypedEventChannelAdmin::TypedEventChannel` interface. When -t is not specified, the event channel object implements the `CosEventChannelAdmin::EventChannel` interface.

When the `destroy()` operation of the event channel is called, the event channel is destroyed and other cleanup functionality may be performed. If -d is passed, a typed event channel also unbinds from the Naming Service and exits the process. Untyped event channels currently do not perform these cleanup tasks.

The -n and -r options are ignored if -x is specified.

**Note** If the -o option is specified without the -x option, the Event Channel object reference is stringified and written to the specified file and bound in the Naming Service.

### 25.6 Event Channel Resource Factory

The event channel resource factory is responsible for creating many strategy objects that control the behavior of the event channel.

The behavior of the event channel is typically controlled by using the service configurator to select the appropriate behaviors for the default factory implementation. Applications are also free to implement their own resource factories, but this is not commonly done. See Chapter 18 for more information on using the service configurator.

The event channel resource factory is registered with the service configurator using the name `CEC_Factory`. The default event channel resource factory is
statically registered with the service configurator, so the `static` directive is used to supply initialization options to it. To change the behavior of the default event channel factory, add a line similar to the line shown below to your service configuration file:

```cpp
static CEC_Factory "-CECDispatching mt -CECDispatchingThreads 5"
```

The option descriptions begin in 25.6.1.1. For these options to be effective, you must make sure that the following function call occurs before the ORB is initialized:

```cpp
TAO_CEC_Default_Factory::init_svcs ()
```

This function creates a default event channel resource factory and statically registers it. If this is not done, the service configurator is not able to find and initialize the `CEC_Factory`.

The `-ORBSvcConf` option allows you to use file names other than `svc.conf` for service configurator initialization. See 19.8.32 for more information on this option.

The default values for many of the event channel resource factory options are defined in `$TAO_ROOT/orbsvcs/orbsvcs/CosEvent/CEC_Defaults.h` as preprocessor macros. You can use your own project-specific defaults by setting these macros in your `config.h` file and recompiling the event service.

## 25.6.1 CEC_Factory Option Overview

This section provides an overview of the configuration options supported by the default `CEC_Factory`. The following section provides detailed documentation of each of the individual options.

### 25.6.1.1 Dispatching

When the event channel is pushing events to interested consumers, choosing the thread used to push the event is a decision that has far-reaching affects on the performance and behavior of the application. The event channel resource factory allows for selection of a dispatching strategy that defines how to push events received from suppliers to the interested consumers on the appropriate thread. The default event channel resource factory allows for either reactive or multithreaded dispatching strategies. In addition, when a multithreaded
dispatching strategy is selected, the number of threads to be used can be specified. Table 25-3 shows the options related to dispatching strategies.

Table 25-3 Dispatching related options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-CECDispatching {reactive</td>
<td>25.6.2.4</td>
<td>Supply this option to select the dispatching strategy for supplier-produced events.</td>
</tr>
<tr>
<td>mt}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-CECDispatchingThreads</td>
<td>25.6.2.5</td>
<td>Specify the number of threads to create and use for the multithreaded dispatching strategy. Defaults to one thread.</td>
</tr>
<tr>
<td>nthreads</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The reactive dispatching strategy delivers events on the same thread as they were received (or generated). This is usually the reactor’s main thread. The mt (multithreaded) dispatching strategy creates a pool of threads and dispatches each event on a randomly-selected member of the pool.

25.6.1.2 Locking Options

The locking options allow the event channel resource factory to define the lock type desired for various components in the event channel. The default factory allows specification of the lock type for consumer and supplier proxies using the options shown in Table 25-4.

Table 25-4 Locking options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-CECProxyConsumerLock {null</td>
<td>25.6.2.7</td>
<td>Specifies the lock type for the consumer proxy object.</td>
</tr>
<tr>
<td>thread</td>
<td>recursive}</td>
<td></td>
</tr>
<tr>
<td>-CECProxySupplierLock {null</td>
<td>25.6.2.10</td>
<td>Specifies the lock type for the supplier proxy object.</td>
</tr>
<tr>
<td>thread</td>
<td>recursive}</td>
<td></td>
</tr>
</tbody>
</table>

These options can be set to null to increase performance if the event channel does not access given components from multiple threads. The default values ensure that the proxy is thread safe, but recursive locks may be required to avoid deadlocks in certain complex systems.

25.6.1.3 Consumer and Supplier Control Options

The following group of options allows the event channel resource factory to define how the event channel handles dangling (ill-behaved) suppliers and consumers. Consumers and suppliers that remain connected to the event
channel when their CORBA objects are no longer accessible from the event channel process are considered ill-behaved. Such consumers and suppliers result when the consumer or supplier process fails to call disconnect, terminates abnormally, or has its node disconnected from the network. The default factory allows specification and configuration of the control policy for consumer and supplier proxies via the options shown in Table 25-5.

Table 25-5 Consumer and supplier control options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-CECConsumerControl (null</td>
<td>reactive)</td>
<td>Define the policy for handling ill-behaved consumers.</td>
</tr>
<tr>
<td>-CECSupplierControl (null</td>
<td>reactive)</td>
<td>Define the policy for handling ill-behaved suppliers.</td>
</tr>
<tr>
<td>-CECConsumerControlPeriod period</td>
<td></td>
<td>Define the polling period in microseconds of the reactive consumer control policy.</td>
</tr>
<tr>
<td>-CECSupplierControlPeriod period</td>
<td></td>
<td>Define the polling period in microseconds of the reactive supplier control policy.</td>
</tr>
<tr>
<td>-CECConsumerControlTimeout timeout</td>
<td></td>
<td>Round-trip timeout in microseconds for the consumer control ping.</td>
</tr>
<tr>
<td>-CECSupplierControlTimeout timeout</td>
<td></td>
<td>Round-trip timeout in microseconds for the supplier control ping.</td>
</tr>
<tr>
<td>-CECProxyDisconnectRetries n</td>
<td></td>
<td>Number of retries allowed for the reactive control strategy.</td>
</tr>
</tbody>
</table>

The default control policy of null leaves consumers and suppliers connected to the event channel even if the event channel is unable to access them. This policy allows consumers and suppliers to continue to be connected even in the face of intermittent communications.

The reactive control policy disconnects a consumer or supplier from the event channel when the event channel fails to contact it. It also periodically polls (by default every 5 seconds) each consumer and supplier to ensure its continued connection. Failure to respond to the polling requests before a timeout (by default 10 milliseconds) also results in disconnection. By default, the first failure to contact a consumer or supplier results in disconnection. You can specify a number of retries using the -CECProxyDisconnectRetries option. The periods for the periodic polls can be set via the -CECSupplierControlPeriod and -CECConsumerControlPeriod
options. If the polling period is set to 0, polling is completely disabled. The round-trip timeout for the periodic polls can be set via the -CECSupplierControlTimeout and -CECConsumerControlTimeout options.

### 25.6.1.4 Proxy Collection Options

The proxy collection options define the types of collections used to hold consumer and supplier proxies. The default factory allows specification of the collection type for consumer and supplier proxies via the options shown in Table 25-6.

**Table 25-6 Proxy collection options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-CECProxyConsumerCollection flags</td>
<td>25.6.2.6</td>
<td>Define the characteristics of the collection used to store proxy consumers in the event channel.</td>
</tr>
<tr>
<td>-CECProxySupplierCollection flags</td>
<td>25.6.2.9</td>
<td>Define the characteristics of the collection used to store proxy suppliers in the event channel.</td>
</tr>
</tbody>
</table>

The flags passed to these collection options fall into three separate groups, with each group specifying a different characteristic of the collection. A colon is used as a separator between flags (e.g., `mt:rb_tree:immediate`).

First, the lock type used to control access to the collection can be specified. The `st` flag allows specification of a null lock. The `mt` flag specifies a thread-safe lock. A thread-safe lock is specified by default.

The second characteristic is the actual collection type used. The `list` flag specifies that an ordered-list collection is used. The `rb_tree` flag specifies that a collection using a red-black tree is used. By default, the event channel uses an ordered-list collection.

The third characteristic specifies how the concurrent use of the collection is controlled, specifically the case where the collection is being iterated over while a client attempts an operation that adds or removes a member of the collection. An example is the distribution of events to push consumers (via iteration over the proxy push suppliers collection) while one of the consumers is attempting to disconnect itself. The default factory provides four different strategies for this characteristic: `immediate`, `copy_on_read`, `copy_on_write`, and `delayed`. 
The *immediate* flag causes each operation to block until it receives access to the collection. In the example above, the consumer that is attempting to disconnect blocks until the event distribution iteration completes. Note that it is possible that the disconnect request may be processed in the same thread as the event distribution (via a nested upcall). If this occurs, immediate access is granted (the thread already has the lock for the collection), and the iterator may be invalidated. It is the developer’s responsibility to ensure that the iterator is not invalidated. Using the `-CECDispatching` option (see 25.6.2.4) to establish a separate dispatching thread is the most common way to ensure this validity. (In other words, the *immediate* collection update flag should not be used with `-CECDispatching reactive`.)

The *copy_on_read* flag causes the iterators to copy the collection before proceeding. This allows iterators to release the lock after the copy is made. Subsequent changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is the extra performance overhead incurred when the copy of the collection is allocated and replicated.

The *copy_on_write* flag causes any modifiers to the collection to make copies of the collection before proceeding. This means changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is the extra performance overhead incurred when the copy of the collection is allocated and replicated. Note that the *copy_on_write* strategy makes a copy each time the collection is changed (connect, disconnect, reconnect, or shutdown), whereas the *copy_on_read* strategy makes copies each time the collection is iterated.

The *delayed* flag causes changes to the collection to be queued while iterations are ongoing. When all iterations have completed, the queued modifications are made. The event channel attributes of *busy_hwm* and *max_write_delay* allow bounds to be set on how many iterators access the collection at a time and how many iterators may access it before modification occurs. See 25.4.2.2 for details of these attributes and how to set them.
25.6.1.5 **Miscellaneous Options**

The options shown in Table 25-7 allow for control of the pull model behavior and the ORB that the event channel uses.

**Table 25-7 Miscellaneous options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-CECReactivePullingPeriod</td>
<td>25.6.2.11</td>
<td>Defines the polling period in microseconds that the reactive pulling strategy uses. The default period is 5000000 (5 seconds).</td>
</tr>
<tr>
<td>-CERCUseORBId orbid</td>
<td>25.6.2.15</td>
<td>Specifies the id of the ORB that the default factory uses.</td>
</tr>
</tbody>
</table>

The reactive pulling period is the period of time between attempted *pulls* on pull suppliers. Events pulled are immediately delivered to push consumers and queued for eventual delivery to pull consumers.

The default factory requires an ORB for a variety of operations. It normally uses the default ORB (with a null string for the ORB id). Specify an ORB id using the -CERCUseORBId option to force the default factory to use a different ORB. Typically, this option is used to ensure that the default factory is using the same ORB as was used to activate the event channel.

25.6.2 **Event Channel Resource Factory Options**

The remainder of this chapter describes the individual options interpreted by the default event channel factory. These options are applied to the default event channel resource factory by the service configurator as described in 25.6.
25.6.2.1 CECConsumerControl *control_policy*

**Values for control_policy**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Do not discard dangling consumers.</td>
</tr>
<tr>
<td>reactive</td>
<td>Use a reactive policy to discard dangling consumers.</td>
</tr>
</tbody>
</table>

**Description**

This option specifies the policy to be used when dealing with dangling consumers. The null control policy never disconnects ill-behaved consumers.

The reactive policy disconnects consumers after a number of communication failures. A communication failure is either a failure to push an event to a consumer or the failure of the periodic ping performed on each consumer. By default, the first communication failure results in the disconnection of the consumer. The number of retries allowed can be set via the -CECProxyDisconnectRetries option.

**Usage**

Use the reactive control policy when consumers could possibly be destroyed without disconnecting. Use the default control policy (null) when you can guarantee that all consumers disconnect properly (or not at all), and you do not want to incur the overhead of the reactive policy.

**Impact**

The null consumer control strategy causes degraded throughput when consumers are destroyed without first disconnecting. The reactive strategy requires slightly more overhead in normal operation, may result in consumers having to reconnect when the network quality is bad (with potential for missed messages), and requires slightly more memory.

**See Also**

25.6.2.2, 25.6.2.3, 25.6.2.8, 25.6.2.12

**Example**

```
static CEC_Factory "-CECConsumerControl reactive"
```
25.6.2.2   CECConsumerControlPeriod *period*

**Description** Sets the period (in microseconds) that the reactive consumer control policy uses to poll the state of the consumers. The default period is 5000000 (5 seconds). A value of zero disables consumer state polling.

**Usage** For event channels using the reactive consumer control policy, use this option to control the time to wait between attempted *pings* on each consumer. The reactive consumer control strategy object pings the consumer by invoking `CORBA::Object::non_existent()` on the consumer’s object reference; this is a synchronous call. The `CECConsumerControlPeriod` option is ignored when the consumer control policy is not reactive.

**Impact** Shorter periods require more bandwidth and processing to validate the existence of the consumers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

**See Also** 25.6.2.1, 25.6.2.13

**Example**

```bash
static CEC_Factory "-CECConsumerControl reactive -CECConsumerControlPeriod 1000000"
```
### 25.6.2.3 CECConsumerControlTimeout `timeout`

**Description**
Sets the relative round-trip timeout (in microseconds) that the reactive consumer control policy uses for polling consumers. The default timeout is 10000 (10 milliseconds).

**Usage**
For event channels using the reactive consumer control policy, use this option to control the time the event channel waits for a consumer to respond to an attempted `ping`. The reactive consumer control strategy object pings the consumer by invoking `CORBA::Object::non_existent()` on the consumer’s object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the event channel classifying that ping as a communication failure for that consumer. The `-CECConsumerControlTimeout` option is ignored when the consumer control policy is not reactive.

**Impact**
Smaller timeout values may result in more timeout failures and consumers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead consumers.

**See Also**
25.6.2.4, 25.6.2.13

**Example**
```
static CEC_Factory "-CECConsumerControl reactive -CECConsumerControlTimeout 50000"
```
**25.6.2.4 CECDispatching dispatching_strategy**

**Values for dispatching_strategy**

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive (default)</td>
<td>The event channel delivers events to consumers on the same thread that received them.</td>
</tr>
<tr>
<td>mt</td>
<td>The event channel uses separate threads to receive and deliver events to consumers. It randomly selects a thread from a pool. The number of threads in the pool is set via the -CECDispatchingThreads option (the default is 1).</td>
</tr>
</tbody>
</table>

**Description**

This option controls the dispatching strategy that the event channel uses. The default strategy is reactive.

**Usage**

This strategy determines the order in which the event channel delivers events as well as its overall throughput. The multithreaded (mt) strategy should allow for greater performance than the reactive model. The reactive strategy is appropriate when it is acceptable and/or desired for all events to be delivered in the order they are received. The mt strategy is especially effective for decoupling the suppliers from the consumers’ execution time, especially in the collocated case. The mt strategy also reduces the maximum latency of event deliveries.

**Impact**

The reactive model is inappropriate when slow processing by individual consumers can affect other users, greater throughput is desired, or the system requires strict prioritization of events. Note that the reactive strategy delivers events on the same thread as they were received. The ORB’s configuration determines the receiving thread. The mt strategy may increase the time required to dispatch an event in lightly-loaded event channels and also require the allocation of additional resources.

**See Also**

21.3.5, 25.6.2.5

**Example**

static CEC_Factory "-CECDispatching mt"
25.6.2.5 **CECDispatchingThreads nthreads**

**Description**  
By default the multithreaded dispatching strategy creates one thread to use for the delivery of supplier-originated events to consumers. Use this option to specify a different number of threads to be created and used.

**Usage**  
Using the multithreaded dispatching strategy with the default of one thread provides the user the benefit of separating the dispatching thread from the receiving thread. This allows for a greater decoupling between suppliers and consumers. Use of this option to specify additional dispatching threads results in additional decoupling between consumers as well as potentially increasing throughput.

**Impact**  
Specifying additional dispatching threads consumes additional resources.

**See Also**  
25.6.2.4

**Example**  
static CEC_Factory "-CECDispatching mt -CECDispatchingThreads 5"
25.6.2.6 CECProxyConsumerCollection flags

Description
This option controls the type of collection the event channel uses to hold consumer proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., `mt:list`). The allowed flags are described in Table 25-8. Only one flag per type should be specified.

Table 25-8 Collection Type Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mt</code> (default)</td>
<td>Synchronization</td>
<td>Use a thread-safe lock for the collection.</td>
</tr>
<tr>
<td><code>st</code></td>
<td></td>
<td>Use a null lock for the collection.</td>
</tr>
<tr>
<td><code>list</code> (default)</td>
<td>Collection</td>
<td>Implement the collection using an ordered list.</td>
</tr>
<tr>
<td><code>rb_tree</code></td>
<td></td>
<td>Implement the collection using a red-black tree.</td>
</tr>
<tr>
<td><code>immediate</code> (default)</td>
<td>Iterator</td>
<td>Threads block until they can execute a change to the collection.</td>
</tr>
<tr>
<td><code>copy_on_read</code></td>
<td></td>
<td>Before initiating an iteration of the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td><code>copy_on_write</code></td>
<td></td>
<td>Before initiating a modification to the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td><code>delayed</code></td>
<td></td>
<td>Changes that cannot be made immediately are queued for later execution.</td>
</tr>
</tbody>
</table>

For a more detailed discussion of the collection types, see 25.6.1.4.

Usage
Applications that guarantee that a consumer proxy collection is only accessed from a single thread can specify the `st` flag to improve performance.

Event channels that connect and disconnect suppliers often, and wish to optimize these operations (at the expense of iteration speed), should specify the `rb_tree` flag.

Applications that use the `immediate` flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. One way to insure that such a collection is not modified is to specify a separate dispatching thread. If you wish to minimize priority inversions between publication and supplier connections and disconnections, use the `delayed` flag. Collections using the `copy_on_read` flag are only applicable to systems with small numbers of consumer proxies that require low latencies for proxy collection modifications. Collections using the `copy_on_write` flag are only applicable to systems with small
numbers of consumer proxies that require low latencies for proxy collection iterations.

**Impact**
The `mt` flag incurs additional overhead over the `st` flag during connection/disconnection of suppliers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of consumer proxy collections (during connection or disconnection of suppliers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Copy on write collections incur dynamic allocation and copy costs for each modification to the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

**See Also** 25.6.2.9

**Example**
```plaintext
static CEC_Factory "-CECProxyConsumerCollection mt:delayed"
```
25.6.2.7 CECProxyConsumerLock *lock_strategy*

**Values for lock_strategy**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Do not use any locking on the proxy consumers.</td>
</tr>
<tr>
<td>thread</td>
<td>Use a thread-safe lock on the proxy consumers.</td>
</tr>
<tr>
<td>recursive</td>
<td>Use a recursive thread-safe lock on the proxy consumers.</td>
</tr>
</tbody>
</table>

**Description**
This option defines the type of lock to be used in synchronizing access to the proxy consumer objects.

**Usage**
Single-threaded applications can use the null lock to increase the efficiency of the push consumer. Multithreaded applications may need to set the lock to recursive in cases where operations on the proxy consumer may cause recursive access to the proxy consumer. In all other situations, the thread lock should be used.

**Impact**
The null lock causes problems in applications that access the proxy from more than one thread. The thread lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The recursive lock is even more expensive than the thread lock, but is required by applications that must recursively access the lock.

**See Also**
25.6.2.10

**Example**
```
static CEC_Factory "-CECProxyConsumerLock recursive"
```
25.6 Event Channel Resource Factory

25.6.2.8 CECProxyDisconnectRetries n

Description  Sets the number of retries that the reactive consumer and supplier control policies use when determining whether to disconnect clients (consumers or suppliers). The default number of retries is zero, meaning that the first failure results in the client being disconnected from the event channel. Each successful communication with the client resets the retry count.

Usage  For event channels using the reactive control policy, use this option to be more tolerant of ill-behaved consumers and suppliers. This option is ignored when the consumer and supplier control policies are not reactive.

Impact  The default value of zero retries means that any failure to contact a consumer or supplier results in their disconnection. This may be too strict for some applications. Larger retry values mean that it takes longer for the event channel to detect and remove ill-behaved clients. This can impact the overall efficiency and performance of the event channel.

See Also  25.6.2.1, 25.6.2.12

Example  static CEC_Factory "-CECConsumerControl reactive -CECProxyDisconnectRetries 3"
25.6.2.9 **CECProxySupplierCollection flags**

**Description**
This option controls the type of collection the event channel uses to hold supplier proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., `mt:list`). The flags are described in Table 25-9. Only one flag per type should be specified.

**Table 25-9 Collection Type Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mt</code> (default)</td>
<td>Synchronization</td>
<td>Use a thread-safe lock for the collection.</td>
</tr>
<tr>
<td><code>st</code></td>
<td></td>
<td>Use a null lock for the collection.</td>
</tr>
<tr>
<td><code>list</code> (default)</td>
<td>Collection</td>
<td>Implement the collection using an ordered list.</td>
</tr>
<tr>
<td><code>rb_tree</code></td>
<td></td>
<td>Implement the collection using a red-black tree.</td>
</tr>
<tr>
<td><code>immediate</code> (default)</td>
<td>Iterator</td>
<td>Threads block until they can execute a change to the collection.</td>
</tr>
<tr>
<td><code>copy_on_read</code></td>
<td></td>
<td>Before initiating an iteration of the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td><code>copy_on_write</code></td>
<td></td>
<td>Before initiating a modification to the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td><code>delayed</code></td>
<td></td>
<td>Changes that cannot be made immediately are queued for later execution.</td>
</tr>
</tbody>
</table>

For a more detailed discussion of the collection types see 25.6.1.4.

**Usage**
Applications that can guarantee that a supplier proxy collection is only accessed from a single thread can specify the `st` flag to improve performance.

Event channels that connect and disconnect consumers often, and wish to optimize these operations (at the expense of iteration speed), should specify the `rb_tree` flag.

Applications that use the `immediate` flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. One way to insure that such a collection is not modified is to specify a separate dispatching thread. If you wish to minimize priority inversions between publication and consumer connections and disconnections, use the `delayed` flag. Collections using the `copy_on_read` flag are only applicable to systems with small numbers of supplier proxies that require low latencies for proxy collection modifications. Collections using the
copy_on_write flag are only applicable to systems with small numbers of supplier proxies that require low latencies for proxy collection iterations.

**Impact**

The mt flag incurs additional overhead over the st flag during connection/disconnection of consumers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of supplier proxy collections (during connection or disconnection of consumers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

**See Also**

25.6.2.6

**Example**

static CEC_Factory "-CECProxySupplierCollection mt:delayed"
25.6.2.10 CECProxySupplierLock *lock_strategy*

### Values for *lock_strategy*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Do not use any locking on the proxy suppliers.</td>
</tr>
<tr>
<td>thread</td>
<td>Use a thread-safe lock on the proxy suppliers.</td>
</tr>
<tr>
<td>recursive</td>
<td>Use a recursive thread-safe lock on the proxy suppliers.</td>
</tr>
</tbody>
</table>

### Description

This option defines the type of lock to use in synchronizing access to the proxy supplier objects.

### Usage

Single-threaded applications can use the *null* lock to increase the efficiency of the push supplier. Multithreaded applications may need to set the lock to *recursive* in cases where operations on the proxy supplier may cause recursive access to the proxy supplier. In all other situations, the *thread* lock should be used.

### Impact

The *null* lock causes problems in applications that access the proxy from more than one thread. The *thread* lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The *recursive* lock is even more expensive than the *thread* lock, but is required by applications that must recursively access the lock.

### See Also

25.6.2.7

### Example

```
static CEC_Factory "-CECProxySupplierLock recursive"
```
25.6.2.11 CECReactivePullingPeriod \textit{period} \\
\textbf{Description} \hspace{1cm} Set the period (in microseconds) the reactive pulling strategy uses to poll all the pull suppliers for events. The default period is 5000000 (5 seconds). \\
\textbf{Usage} \hspace{1cm} The reactive pulling strategy periodically attempts to pull events from each pull supplier attached to an event channel. This option allows applications to customize the period between attempted pulls. These periodic attempts are made using the reactor associated with the default factory's ORB. These pull requests use a default timeout value of ten milliseconds. Alternate timeout values are specified via the \texttt{-CECSupplierControlTimeout} option. \\
\textbf{Impact} \hspace{1cm} Shorter periods incur greater bandwidth and processing demands. Longer periods cause greater delays between the availability of events and their eventual delivery. \\
\textbf{See Also} \hspace{1cm} 25.6.2.15, 25.6.2.14 \\
\textbf{Example} \hspace{1cm} static CEC\_Factory "-CECReactivePullingPeriod 1000000"
25.6.2.12 CECSupplierControl control_policy

Values for control_policy

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Do not discard dangling suppliers.</td>
</tr>
<tr>
<td>reactive</td>
<td>Use a reactive policy to discard dangling suppliers.</td>
</tr>
</tbody>
</table>

Description

This option specifies the policy used when dealing with dangling suppliers. The null control policy never disconnects ill-behaved suppliers.

The reactive policy disconnects suppliers after a number of communication failures. A communication failure is either the failure of the periodic ping performed on the supplier or any other failure to correctly invoke an operation on the supplier. By default, the first communication failure results in the disconnection of the supplier. The number of retries allowed can be set via the -CECProxyDisconnectRetries option.

Usage

Use the reactive control policy when suppliers could possibly be destroyed without disconnecting. Use the default control policy (null) when you can guarantee that all suppliers disconnect properly (or not at all), and you do not want to incur the overhead of the reactive policy.

Impact

The null supplier control strategy causes degraded throughput when suppliers are destroyed without first disconnecting. The reactive strategy requires slightly more overhead in normal operation, may result in suppliers having to reconnect when the network quality is bad, and requires slightly more memory.

See Also

25.6.2.1, 25.6.2.13, 25.6.2.14, 25.6.2.8

Example

static CEC_Factory "-CECSupplierControl reactive"
25.6.2.13 CECSupplierControlPeriod *period*

**Description**  Sets the period (in microseconds) that the reactive supplier control policy uses to poll the state of the suppliers. The default period is 5000000 (5 seconds). A value of zero disables supplier state polling.

**Usage**  For event channels using the reactive supplier control policy, use this option to control the time to wait between attempted *pings* on each supplier. The reactive supplier control strategy object pings the supplier by invoking `CORBA::Object::_non_existent()` on the supplier’s object reference; this is a synchronous call. The `-CECSupplierControlPeriod` option is ignored when the consumer control policy is not reactive.

**Impact**  Shorter periods require more bandwidth and processing to validate the existence of the suppliers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

**See Also**  25.6.2.2, 25.6.2.12

**Example**  
```
static CEC_Factory "-CECSupplierControl reactive -CECSupplierControlPeriod 1000000"
```
25.6.2.14 CECSupplierControlTimeout timeout

Description  Sets the relative round-trip timeout (in microseconds) used for polling suppliers. This timeout is used both for polling pull model suppliers and by the reactive supplier control strategy. The default timeout is 10000 (10 milliseconds).

Usage  For event channels using the reactive supplier control policy, use this option to control the time the event channel waits for a supplier to respond to an attempted ping. For pull suppliers, use this option to control the time the event channel waits for each pull() call to a pull supplier. The reactive supplier control strategy object pings the supplier by invoking CORBA::Object::_non_existent() on the supplier’s object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the event channel classifying that ping as a communication failure for that supplier.

Impact  Smaller timeout values may result in more timeout failures and suppliers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead suppliers.

See Also  25.6.2.12, 25.6.2.3

Example  static CEC_Factory "-CECSupplierControl reactive -CECSupplierControlTimeout 50000"
25.6.2.15 CECUseORBId *orb-id*

**Description**  
Sets the name of the ORB used by the default factory implementation. The default factory creates strategy objects that use this ORB to perform remote invocations and to gain access to the ORB’s reactor.

**Usage**  
This option is only useful in applications that create multiple ORBs and activate the event channel in one of them. Use it to ensure that the objects created by the default factory use the same ORB as the event channel and related objects.

**Impact**  
This option may cause the creation of a new ORB (and associated resources), if the ORB with the given name has not been initialized.

**Example**  
static CEC_Factory "-CECUseORBId Orb2"
26.1 Introduction

Though the OMG Event Service defines the basic interfaces necessary for decoupling suppliers and consumers of data, it leaves a number of important details out, particularly in areas affecting real-time systems. The TAO Real-Time Event Service (RTES) addresses a number of these issues by adding support for features such as event filtering, event correlation, real-time event dispatching/scheduling, and periodic event processing. Many non-real-time applications also benefit from using the RTES as a fast and predictable event distribution mechanism with additional features not included in the OMG Event Service.

26.2 Overview of the TAO Real-Time Event Service

The TAO Real-Time Event Service’s interfaces for publishing and receiving events remain mostly the same as in the standard OMG Event Service. The programmer must provide additional information when creating an event channel and when registering suppliers and consumers to allow the TAO
Real-Time Event Service to function as desired. The Real-Time Event Service only supports the push model of operation.

Figure 26-1 shows the basic architecture of the TAO Real-Time Event Service:

![Real-Time Event Service Architecture](image)

**Figure 26-1 Real-Time Event Service Architecture**

Each element of the Real-Time Event Service architecture is described below.

### 26.2.1 Consumer Proxies

This module implements a `SupplierAdmin` interface analogous to that of the OMG Event Service. Suppliers use this interface to create objects that support
the ProxyPushConsumer interface. Suppliers use the ProxyPushConsumer interface to connect and disconnect from the event channel as well as publish events via the push() operation.

### 26.2.2 Priority Timers
The RTES allows consumers to define timeout events that occur when a desired duration has elapsed. These timeout events can be made dependent upon the occurrence of other events. This module generates and manages these timeout events.

### 26.2.3 Subscription and Filtering
This module handles distribution of events to the consumers. TAO adds support for event filtering on consumers based on the source and/or type of the event published.

### 26.2.4 Event Correlation
This module correlates events based on details the consumer provides. The consumer may request to only receive particular events when some other related event has also occurred. In this case the Event Correlation module holds the first event and delivers them both when the second is published.

### 26.2.5 Dispatching Module
This module determines when to deliver events based on a variety of information the user provides.

### 26.2.6 Supplier Proxies
This module implements the ConsumerAdmin interface analogous to that of the OMG Event Service. Consumers use this interface to create objects that support the ProxyPushSupplier interface. The ProxyPushSupplier interface is used to connect and disconnect from the event channel. TAO extends this interface to allow the consumer to indicate the dependencies and other details the Real-Time Event Service needs.

### 26.2.7 Real-Time Event Service Libraries
The RTES functionality is split into the libraries defined in Table 26-1. The majority of the functionality is in the TAO_RTEvent library. All examples and
features in this section require this library to be linked with your application. When other libraries are required we will explicitly state the library required.

Table 26-1 Real-Time Event Service Libraries

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAO_RTEvent</td>
<td>Basic RT Event Service functionality.</td>
</tr>
<tr>
<td>TAO_RTOLDEvent</td>
<td>“Old” RT Event Service implementation. For use by legacy applications.</td>
</tr>
<tr>
<td>TAO_RTKokyuEvent</td>
<td>Allows integration of RTES with the Kokyu Scheduling Framework.</td>
</tr>
<tr>
<td>TAO_RTSchedEvent</td>
<td>Allows integration of RTES with TAO’s original scheduling service.</td>
</tr>
<tr>
<td>TAO_RTCORBAEvent</td>
<td>Allows use of the Real-Time CORBA thread pools for RTES dispatching.</td>
</tr>
</tbody>
</table>

Note: There are two implementations of real-time event channels in TAO. The old, deprecated implementation is defined in the ACE_EventChannel class and is included in the TAO_RTOLDEvent library. The new and improved implementation is in the TAO_EC_Event_Channel class and is included in the TAO_RTEvent library. The old implementation only remains to support existing applications as they transition to the new implementation. The two implementations should be functionally equivalent at the interface level as they share the same IDL, but many details differ including their configuration and the functionality of specific features. This chapter primarily discusses the new implementation and always refers to the old event channel explicitly when it is being discussed.

26.3 Using the TAO Real-Time Event Service

The examples in this chapter explore some of the features that the TAO Real-Time Event Service introduces. The Naming Service server must be started prior to running these examples and the other processes must be configured to use the same naming server (either via multicast, the NameServiceIOR environment variable, or the -ORBInitRef option).

26.3.1 A Basic Example

This example is functionally the same as the basic example in Chapter 25. The main differences are accounted for by the fact that the Real-Time Event
Service defines interfaces for the Event Channel, Supplier, Consumer, and Admin concepts that are analogous to the OMG Event Service interfaces, but the RTES interfaces take slightly different parameters than the standard event service equivalents. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/EventServices/RTEC_Basic.

26.3.1.1 Starting the Event_Service Process

This example utilizes the Event_Service server that is delivered with TAO. This server creates a single RTES event channel object and binds it in the root naming context of the Naming Service. By default, it binds to the name “EventService”. The options that can be passed to this server are covered in 26.4. A typical invocation of the server is:

$TAO_ROOT/orbsvcs/Event_Service/Event_Service

This starts an Event Channel with a local Scheduler and binds both in the root naming context of the Naming Service. See the Scheduling Service documentation for additional information regarding the scheduler.

26.3.1.2 The Supplier

To use the RTES, your supplier must include the header files that the IDL compiler generated from the Real-Time Event Service’s IDL. This IDL file defines interfaces that are very similar to those defined by the standard OMG Event Service. To differentiate the interfaces, the module and file names of TAO’s Real-Time Event Service are given the Rtec prefix. Listed below are the include directives for EchoEventSupplierMain.cpp.

```c++
#include <orbsvcs/RtecEventCommC.h>
#include <orbsvcs/RtecEventChannelAdminC.h>
```

After initializing the ORB and resolving and narrowing the Naming Service’s root context, this context is used to retrieve an object reference to the event channel using the name “EventService”. This event channel reference is used to get a supplier administration object reference that is then used to create a consumer proxy. All of this works as in the standard OMG Event Service, except for the Rtec prefix on the modules/namespaces in the source code.

```c++
// Get the Event Channel using the Naming Service
```
Next, create the supplier servant. In the OMG Event Service based example, passing a null reference to the proxy consumer let us avoid creating a supplier servant. The RTES always requires a supplier CORBA object that implements RtecEventComm::PushSupplier when connecting to a proxy consumer. Only the disconnect_push_supplier() operation is required for this object, although it is often given many other application-specific responsibilities.

To connect to the event channel using the Real-Time Event Service, the supplier must provide some quality of service (QoS) information. This is created using a QoS factory object. Information about the events that this supplier publishes are inserted into the QoS factory, which builds up an internal data structure that is later passed to connect_push_supplier().

const RtecEventComm::EventSourceID MY_SOURCE_ID = ACE_ES_EVENT_SOURCE_ANY + 1;
26.3 Using the TAO Real-Time Event Service

const RtecEventComm::EventType MY_EVENT_TYPE = ACE_ES_EVENT_UNDEFINED + 1;

// Publish the events the supplier provides.
ACE_SupplierQOS_Factory qos;
qos.insert (MY_SOURCE_ID, // Source ID
            MY_EVENT_TYPE, // Event type
            0,             // handle to the rt_info structure
            1);            // number of calls

// Connect as a supplier of the published events.
consumer->connect_push_supplier (supplier.in (),
                                 qos.get_SupplierQOS ());

When inserting the QoS data, the source identifier must be a unique non-zero long integer for each supplier in the system. User-defined event types must start at a number higher than the value defined by the preprocessor macro ACE_ES_EVENT_UNDEFINED. Passing the preprocessor macro ACE_ES_EVENT_ANY as the event type allows this supplier to publish events of any type.

Passing zero as the handle to the real-time information (rt_info) data structure tells the event channel to just deliver the events in the order received. The rt_info data structure is the main mechanism for controlling the real-time capabilities of the real-time Event Service. See the appropriate Scheduling Service documentation for more details.

Once the supplier is connected, the event data can be created. Note that the RTES-based interfaces publish sets of events and that these events have more fields than the OMG Event Service’s events. Here an EventSet is created with one event and a payload of a string stored in the any_value member:

CORBA::String_var eventData = CORBA::string_dup("Hello, world.");

// Create an event set for one event
RtecEventComm::EventSet events (1);
events.length (1);

// Initialize event header.
events[0].header.source = MY_SOURCE_ID;
events[0].header.type = MY_EVENT_TYPE;

// Initialize data fields in event.
events[0].data.any_value <<= eventData;
Publishing the events is done using the `push()` operation on the proxy consumer interface:

```cpp
caller->push (events);
```

### 26.3.1.3 Implementing the Push Consumer Interface

To create a consumer, the `RtecEventComm::PushConsumer` IDL interface must be implemented. There are only two operations in this interface, `push()` and `disconnect_push_consumer()` as shown in the file `EchoEventConsumer_i.h` for this example.

```cpp
#include <orbsvcs/RtecEventCommS.h>

class EchoEventConsumer_i : public virtual POA_RtecEventComm::PushConsumer
{
  public:
    // Constructor
    EchoEventConsumer_i(CORBA::ORB_ptr orb);

    // Override operations from PushConsumer interface.
    virtual void push(const RtecEventComm::EventSet& events)
      throw(CORBA::SystemException);

    virtual void disconnect_push_consumer() throw(CORBA::SystemException);

  private:
    CORBA::ORB_var orb_;
};
```

The implementations for these operations are contained in the file `EchoEventConsumer_i.cpp`. The `push()` operation implementation is the only code that significantly differs from an OMG Event Service push consumer:

```cpp
// Implement the push() operation.
void EchoEventConsumer_i::push(const RtecEventComm::EventSet& events)
  throw(CORBA::SystemException);
{
  for (u_int i = 0; i < events.length (); ++i) {
    // Extract event data from the any.
    const char* eventData;
    std::cout << "Received event,"
    << " type: " << events[i].header.type
    << " source: " << events[i].header.source;
```
26.3 Using the TAO Real-Time Event Service

```cpp
if (events[i].data.any_value >>= eventData) {
    std::cout << "  text: "   << eventData;
}
std::cout << std::endl;
}
```

The first difference is that the `push()` operation must now process a set of events instead of a single event. Secondly, instead of a simple `CORBA::Any` parameter, the Real-Time Event Service’s `Event` data structure has significantly more fields. In this case, the supplier has used the `any_value` field embedded in the `Event`’s body to pass a string to the consumer. The consumer also extracts the `type` and `source` fields from the header portion of the `Event` data structure.

26.3.1.4 Creating the Consumer and Connecting to the Channel

The consumer application accesses the event channel using the same code as shown previously for the supplier (refer to `EchoEventConsumerMain.cpp` for this example). The event channel is then used to obtain a proxy push supplier:

```cpp
// Obtain a reference to the consumer administration object.
RtecEventChannelAdmin::ConsumerAdmin_var admin = ec->for_consumers();

// Obtain a reference to the push supplier proxy.
RtecEventChannelAdmin::ProxyPushSupplier_var supplier =
    admin->obtain_push_supplier();
```

Now an instance of the consumer servant can be created and connected to the event channel using QoS information similar to that seen on the supplier side. The QoS is created using a consumer QoS factory and the only information required is an event type and a handle to the real-time information data structure. This example uses as the event type `ACE_ES_EVENT_ANY`, which is a wildcard that accepts all event types. It passes 0 for the handle to the real-time information, causing the event channel to deliver the events in the order received.

```cpp
// Get the RootPOA.
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

// Instantiate an EchoEventConsumer_i servant.
```
EchoEventConsumer_i servant(orb.in());

// Register it with the RootPOA.
PortableServer::ObjectId_var oid = poa->activate_object(&servant);
CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
RtecEventComm::PushConsumer_var consumer =
    RtecEventComm::PushConsumer::_narrow(consumer_obj.in());

// Connect as a consumer.
ACE_ConsumerQOS_Factory qos;
quos.start_disjunction_group (1);
quos.insert_type (ACE_ES_EVENT_ANY, // Event Type
    0); // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
    qos.get_ConsumerQOS ());

After activating the POA manager and starting the event loop the application
is ready to receive events.

// Activate the POA via its POAManager.
PortableServer::POAManager_var poa_manager = poa->the_POAManager();
poa_manager->activate();

std::cout << "EchoEventConsumerMain.cpp: Ready to receive events..."
    << std::endl;

// Enter the ORB event loop.
orb->run();

// If we have reached this, we must be shutting down...
// Disconnect the ProxyPushSupplier.
supplier->disconnect_push_supplier();
supplier = RtecEventChannelAdmin::ProxyPushSupplier::_nil();
admin = RtecEventChannelAdmin::ConsumerAdmin::_nil();

orb->destroy();

All events from the event channel are now delivered to the consumer object’s
push() operation.

26.3.2 Managing Connections
The real-time event service interfaces provide operations for consumers and
suppliers to manage their connections and control the delivery of events.
26.3.2.1 Connecting and Disconnecting Consumers

The previous example connected its consumer using the `connect_push_consumer()` operation on the `ProxyPushSupplier` interface:

```
supplier_proxy->connect_push_consumer (consumer.in (),
  qos.get_ConsumerQOS ());
```

Once this has executed, the consumer remains connected to the event channel until the corresponding disconnect operation is called:

```
supplier_proxy->disconnect_push_supplier ();
```

This disconnects the consumer from the event channel. Failure to disconnect from the event channel before ending the process or destroying the consumer leaves references to the consumer in the event channel. By default, the event channel continues to attempt to push events to the consumer, but fails each time. See 26.5.1.4 for details on how to configure the real-time event channel so as to ensure automatic removal of such “dangling” consumer object references.

26.3.2.2 Connecting and Disconnecting Suppliers

Suppliers are connected and disconnected using operations on the proxy push consumer that are analogous to operations on the proxy push supplier:

```
consumer_proxy->connect_push_supplier (supplier.in (),
  qos.get_SupplierQOS ());

// Do all the publication required...

consumer_proxy->disconnect_push_consumer ();
```

Failure to disconnect from the event channel leads to similar problems in the supplier case. Instead of affecting each push though, it generally affects the speed of connection and disconnection by consumers. See 26.5.1.4 for details on how to configure the real-time event channel so as to ensure automatic removal of “dangling” supplier object references.
26.3.2.3 Reconnecting Consumers and Suppliers

To change details of their connection (such as filter information or timeouts), consumers and suppliers can invoke the appropriate connect operation even after a connection is established. This is more efficient than explicitly disconnecting and reconnecting. For example, consider the following:

```c++
supplier_proxy->connect_push_consumer (consumer.in (),
    qos.get_ConsumerQOS ());

// Process events for a while...

supplier_proxy->connect_push_consumer (consumer.in (),
    qos2.get_ConsumerQOS ());
```

If we assume that `qos` and `qos2` refer to different subscription details such as filtering for different event types then this code changes the subscription while remaining connected to the event channel. No events should be lost during this change.

**Note**

To support this feature the event channel must have the attributes `consumer_reconnect` and `supplier_reconnect` set to true. See 26.3.6.2 for details on setting these attributes. Attempting to reconnect if these attributes are not set to true results in the `connect_ * ()` operation throwing an `RtecEventChannelAdmin::AlreadyConnected` exception.

26.3.2.4 Suspending and Resuming Consumer Connections

If a consumer wants to stop processing events for a time, it could remain connected to the event channel and simply ignore events as they arrive. However, this wastes network bandwidth and processor resources. A more efficient alternative is to temporarily suspend the connection, then later resume the connection. To suspend the consumer’s connection to the event channel, use the `suspend_connection() ` operation, which causes the event channel to stop pushing events to the consumer. You can later resume the connection and again receive events by invoking the `resume_connection() ` operation.

Another approach is to disconnect from the event channel and reconnect. However, the `suspend_connection() ` and `resume_connection() ` operations are more efficient because they leave the subscription in place (and thereby avoid locking the proxy collection so it can be changed). This allows
the event channel to resume the subscription much faster than establishing a new subscription.

supplier_proxy->suspend_connection();

// Do something else for a while...

supplier_proxy->resume_connection();

The `suspend_connection()` operation sets a flag in the proxy supplier of the event channel that blocks all publications until the `resume_connection()` operation is called and resets the flag. Any events published while the connection is suspended are lost to that consumer.

There is a slight overhead involved in keeping the suspended subscriptions in the event channel that may make it desirable to disconnect when doing so for long periods of time. The suspend/resume mechanism is an attractive solution for brief suspensions.

### 26.3.3 The Event Structure

The TAO Real-Time Event Service uses event structures that are defined in `$TAO_ROOT/orbsvcs/orbsvcs/RtecEventComm.idl` in the `RtecEventComm` module. Here are the basic definitions of `EventSet` and `Event`:

```c
struct Event
{
    EventHeader header;
    EventData data;
};
typedef sequence<Event> EventSet;
```

#### 26.3.3.1 The EventHeader Structure

Table 26-2 describes the members of the `EventHeader` structure.

<table>
<thead>
<tr>
<th>Type (typedef type)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventType (long)</td>
<td>type</td>
<td>User-defined type field.</td>
</tr>
<tr>
<td>EventSourceID (long)</td>
<td>source</td>
<td>User-defined source id.</td>
</tr>
</tbody>
</table>
Table 26-2 EventHeader Members

<table>
<thead>
<tr>
<th>Type (typedef type)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>ttl</td>
<td>Time to live count for federations/gateways.</td>
</tr>
<tr>
<td>Time (TimeBase::TimeT)</td>
<td>creation_time</td>
<td>User-defined timing information.</td>
</tr>
<tr>
<td>Time (TimeBase::TimeT)</td>
<td>ec_recv_time</td>
<td>User-defined timing information.</td>
</tr>
<tr>
<td>Time (TimeBase::TimeT)</td>
<td>ec_send_time</td>
<td>User-defined timing information.</td>
</tr>
</tbody>
</table>

Setting the **type** and **source** members is generally sufficient for most simple applications.

The **ttl** member is a counter that is decremented each time an event is passed between individual event channels in a federation (see 26.3.8 for details). When the counter reaches zero, the event is no longer passed to additional event channels in the federation.

Table 26-3 lists special values that are defined for **EventTypes**:

**Table 26-3 EventType Special Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE_ES_EVENT_ANY</td>
<td>Wild card value that matches all event types.</td>
</tr>
<tr>
<td>ACE_ES_EVENT_SHUTDOWN</td>
<td>Event type that can be used for shutdown events from the supplier.</td>
</tr>
<tr>
<td>ACE_ES_EVENT_INTERVAL_TIMEOUT</td>
<td>Event type for interval timeout events.</td>
</tr>
<tr>
<td>ACE_ES_EVENT_DEADLINE_TIMEOUT</td>
<td>Event type for deadline timeout events.</td>
</tr>
<tr>
<td>ACE_ES_EVENT_UNDEFINED</td>
<td>This value marks the beginning of the range of user-defined event types.</td>
</tr>
</tbody>
</table>

There are several other special values defined for internal use, all of which are values smaller than the value of the **ACE_ES_EVENT_UNDEFINED** preprocessor macro.

#### 26.3.3.2 The EventData Structure

Table 26-4 describes the members of the **EventData** structure:

**Table 26-4 EventData Members**

<table>
<thead>
<tr>
<th>Type (typedef type)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>pad1</td>
<td>Pad field, improves alignment and performance.</td>
</tr>
</tbody>
</table>
26.3 Using the TAO Real-Time Event Service

The choice of whether to use the any_value or the payload member in a particular application is a design decision, based upon the type of application and the nature of the event data. In many situations, custom marshaling and demarshaling of the EventPayload may be more efficient than using a CORBA::Any.

### 26.3.3.3 Customizing the Event Structure

TAO provides several build options that allow removal of unneeded members from the event structure. Table 26-5 describes these options.

**Table 26-5 Event Structure Build Options**

<table>
<thead>
<tr>
<th>Build Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAO_LACKS_EVENT_CHANNEL_ANY</td>
<td>Removes the any_value member from the EventData structure.</td>
</tr>
<tr>
<td>TAO_LACKS_EVENT_CHANNEL_OCTET_SEQUENCE</td>
<td>Removes the payload member from the EventData structure.</td>
</tr>
<tr>
<td>TAO_LACKS_EVENT_CHANNEL_TIMESTAMPS</td>
<td>Removes the ec_recv_time and ec_send_time members from the EventHeader structure.</td>
</tr>
</tbody>
</table>

To take advantage of these optimizations, the TAO_RTEvent library and any executables using it (e.g., Event_Service) must be rebuilt. The flags can be set by adding them to the platform_macros.GNU file, uncommenting the appropriate lines in $TAO_ROOT/rules.tao.GNU, or passing them to make via the MAKEFLAGS environment variable or the command line. See Appendix A for more details on build flags and their use.

In addition, users can add their own members to the EventData structure in the file $TAO_ROOT/orbsvcs/orbsvcs/RtecDefaultEventData.idl. Once again, this requires that the TAO_RTEvent library and associated executables be rebuilt. To interoperate using the event channel, all processes must use the same event structure.

<table>
<thead>
<tr>
<th>Type (typedef type)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventPayload(sequence&lt;octet&gt;)</td>
<td>payload</td>
<td>User-defined payload using a sequence of octet. Users must supply their own marshaling.</td>
</tr>
<tr>
<td>any</td>
<td>any_value</td>
<td>User-defined payload using a CORBA::Any type.</td>
</tr>
</tbody>
</table>
26.3.4 Filtering and Correlation

The TAO Real-Time Event Service gives consumers the capability to filter the events delivered over the event channel to specific consumers. Events can be filtered based on the event type and source id, as well as combinations of the type and source.

The above elements can also be combined in groups with a variety of semantics. Conjunction groups give the consumer the ability to delay the delivery of some events until all events in the group are available. Events that match the different elements of a conjunction group are queued until all elements of the group have been matched. Filters can also be combined in disjunction groups that deliver events every time any of the events in the group are available. Logical AND groups require events to pass all filters in the group before they are delivered.

Conjunction, disjunction, and logical AND groups can also be nested within one another to create more complex filters (although the event channel may be configured so as to limit this nesting).

Negations allow individual filters (including groups) to have their logic inverted so that only events that fail a particular filter are passed to the consumer. The event channel architecture also supports bit-mask filters that enable faster filtering based on bit masks.

Full source code for this example is in the TAO 1.4a source code distribution in the directory


26.3.4.1 Specifying and Constructing Filters

Filters are specified when connecting a consumer to the event channel via the connect_push_consumer() operation. The previous examples use an ACE_ConsumerQOS_Factory to construct the ConsumerQOS structure that was then passed to connect_push_consumer().

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_type (ACE_ES_EVENT_ANY,  // Event Type
0);                // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
qos.get_ConsumerQOS ());
```

Here is the definition of the ConsumerQOS structure in IDL:
module RtecEventChannelAdmin {
    struct Dependency { 
        RtecEventComm::Event event;
        RtecScheduler::handle_t rt_info;
    };
    typedef sequence<Dependency> DependencySet;
    struct ConsumerQOS {
        DependencySet dependencies;
        boolean is_gateway;
    };
    // Rest of the module...
};

The core of this structure is a sequence of events. Special event types are used to specify the different filter types. In 26.3.4.13, we discuss how to build custom filters.

When the ConsumerQOS is passed to the event channel, the event channel uses a filter builder to construct a tree of filter objects that is used to filter subsequent events. Each time an event is pushed, the event channel evaluates that event using the filter tree for each consumer. The tree of filters causes some events to be discarded (never delivered to that consumer), some to be queued (for potential later delivery to that consumer), and some to be immediately delivered.

The type of filter builder that the event channel uses is specified via the -ECFFiltering option (see 26.5.2.5) and determines how the ConsumerQOS structure is translated into the filter object tree. For example, a null filter builder ignores the ConsumerQOS structure and builds a filter tree that passes all events. A basic filter builder supports all the basic types, but limits the nesting of filter groups (conjunction, disjunction, logical AND) to two levels. The prefix filter builder allows arbitrarily deep nesting, but requires additional information in the subscription to do so.

### 26.3.4.2 Filtering By Event Type

In the basic RTES example in 26.3.1, the consumer uses the special event type of ACE_ES_EVENT_ANY to accept events of all types:

```cpp
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_type (ACE_ES_EVENT_ANY, // Event Type
    0); // handle to the rt_info
```
Real-Time Event Service

By calling `insert_type()` with specific event types, consumers can elect to receive only events of those particular types:

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_type (MY_EVENT_TYPE, 0); // Event Type
supplier->connect_push_consumer (consumer.in (),
        qos.get_ConsumerQOS ());
```

This consumer receives all events whose type field has a value of `MY_EVENT_TYPE`. All other events are not to be delivered to this consumer.

26.3.4.3 Filtering By Source ID

In the previous examples, no source IDs were specified in the QoS information so events from all sources are delivered by default. By calling `insert_source()`, the consumer can limit the delivery of events to those originating from a particular source or sources.

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_source (MY_SOURCE_ID, 0); // Source ID
supplier->connect_push_consumer (consumer.in (),
        qos.get_ConsumerQOS ());
```

The event channel now delivers to this consumer only those events whose source id field matches `MY_SOURCE_ID`.

26.3.4.4 Filtering By Source ID and Event Type Combinations

The consumer QoS factory also allows filtering based on specific Source Id/Event Type combinations via `insert()`.

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert (MY_SOURCE_ID, MY_EVENT_TYPE, 0); // Source Id/Event Type
supplier->connect_push_consumer (consumer.in (),
        qos.get_ConsumerQOS ());
```
This causes all events of type `MY_EVENT_TYPE` with a source of `MY_SOURCE_ID` to be delivered to this consumer. All other combinations of event types and source identifiers are ignored.

---

**Note**

> When calling `insert()`, be sure to pass the source identifier first and the event type second as these are easily swapped and the compiler does not provide a warning.

### 26.3.4.5 Bit-Mask Filters

The bit-mask value filter allows users to apply (bitwise AND) bit masks to the source and event type fields and attempts to match the result with specified values. Here is an example of a consumer with a bit-mask value filter:

```cpp
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_bitmasked_value (0xFFFFFFFF, // Source mask
0x0000FFFF, // Type mask
0x00000020, // Source value
0x00000000); // Type value
supplier->connect_push_consumer (consumer.in (),
qos.get_ConsumerQOS());
```

This code results in a filter that performs a bitwise AND of the source mask (`0xFFFFFFFF`) and the source id of the event and compares the result with source value passed (`0x00000020`). It also does a bitwise AND of the type mask (`0x0000FFFF`) and the type of the event and compares the result with the type value passed (`0x00000000`). The end result is a filter that passes all events with a source id of 32 (`0x00000020`) and an event type with none of the lower 16 bits set.

### 26.3.4.6 Disjunction Groups

So far, the examples in this chapter have utilized a disjunction group that contains a single filter. Invocations of `insert_type()`, `insert_source()`, and `insert()` create filters of the appropriate types. Each disjunction group can contain an unlimited number of these filters.

Disjunction groups cause the event channel to deliver an individual event whenever any of the filters contained within the group “match” the event. To
start a disjunction group, invoke `start_disjunction_group()` and pass it the number of child filters that it will contain. The number of child filters argument can be omitted if the basic filter builder is used (see 26.3.4.10 for details).

Here is an example of a more complex disjunction group.

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (5);
qos.insert_type (MY_EVENT_1, // Event Type
                   0); // handle to the rt_info
qos.insert_type (MY_EVENT_2, // Event Type
                   0); // handle to the rt_info
qos.insert_source (MY_SOURCE_1, // Source ID
                   0); // handle to the rt_info
qos.insert_source (MY_SOURCE_2, // Source ID
                   0); // handle to the rt_info
qos.insert (MY_SOURCE_3,        // Source ID
            MY_EVENT_3,          // Event Type
            0);                  // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                  qos.get_ConsumerQOS ());
```

This consumer receives all events of type `MY_EVENT_1` and `MY_EVENT_2`, all events from `MY_SOURCE_1` and `MY_SOURCE_2`, and all events of type `MY_EVENT_3` from `MY_SOURCE_3`. Each of these events is delivered as it is pushed to the event channel.

### 26.3.4.7 Conjunction Groups

Construction of conjunction groups is accomplished in a similar manner:

```c
ACE_ConsumerQOS_Factory qos;
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_1, // Event Type
                 0); // handle to the rt_info
qos.insert_type (MY_EVENT_2, // Event Type
                 0); // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                  qos.get_ConsumerQOS ());
```

Instead of delivering each matching event as it is received, the conjunction group causes the event channel to queue events until an event matching each filter in the conjunction is available. At that point, the event channel delivers all events in the group to the consumer (in the same EventSet). In the above
example, if the event channel first receives an event of type \texttt{MY\_EVENT\_1}, the event channel will queue that event until an event of type \texttt{MY\_EVENT\_2} is received. At that point, the event channel will deliver both events to the consumer via one call to the consumer’s \texttt{push()} operation.

This feature gives the consumer the ability to correlate related events and process them as a group. In some applications, this may increase the performance of the system and reduce the workload for the consumer.

The conjunction only queues one event for each corresponding filter in the group. Any subsequent events that match that filter are discarded until the conjunction is satisfied and its events delivered. In the above example, this means that once a \texttt{MY\_EVENT\_1} type event is queued in the event channel, all subsequent events of this type are discarded until a \texttt{MY\_EVENT\_2} event is received and the conjunction is satisfied.

### 26.3.4.8 Logical AND Groups

Construction of logical AND groups is done in a similar manner to disjunction and conjunction groups:

```c
ACE_ConsumerQOS.Factory qos;
qos.start_logical_and_group (2);
// This bitmask matches events from any source with none of the high 16 bits of // the event type set
qos.insert_bitmasked_value (0x00000000, // Source mask
    0xFFFF0000,   // Type mask
    0x00000000,   // Source value
    0x00000000);  // Type value
qos.insert_source (MY\_SOURCE,    // Source ID
    0);           // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
    qos.get_ConsumerQOS ());
```

This code causes the consumer to only receive events that pass the source and bit-mask value filters specified (it passes all events from \texttt{MY\_SOURCE} with none of the high 16 bits set on the event type).

### 26.3.4.9 Negating the Logic of Filters

Filter logic can be inverted with the use of negations. If a given tree of filters would normally pass one group of events and filter another, then negation of that tree of filters would pass the events formerly filtered and filter the ones formerly passed. Negations can only directly contain a single filter (although
that filter may be a group that contains other filters). Here is an example of a negation filter.

```c
ACE_ConsumerQOS_Factory qos;
qos.start_negation ();
qos.insert_source (MY_SOURCE, // Source ID
                  0);           // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                  qos.get_ConsumerQOS ());
```

This consumer receives events from any source other than `MY_SOURCE`.

### 26.3.4.10 Nesting Groups

The TAO Real-Time Event Service supports nesting of conjunction, disjunction, logical AND, and negation groups to arbitrarily deep levels. The default filter builder (basic) only supports two levels of nesting with the top level being restricted to an implicit disjunction group. The start of one group terminates the previous group. Here is an example:

```c
ACE_ConsumerQOS_Factory qos;
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_type (MY_EVENT_2, 0);
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_3, 0);
qos.insert_type (MY_EVENT_4, 0);
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This code creates two conjunction groups nested inside of the implicit disjunction group. This consumer is delivered pairs of events (types 1 and 2 together and types 3 and 4 together) as the event channel collects the appropriate matching events.

Because the basic filter builder assumes that the start of a subsequent group means the end of the previous one, the arguments to the functions that start groups can be omitted. In this case, the filter builder calculates the number of children. However, we strongly recommend the explicit use of the arguments to the start methods. If they are omitted and the prefix filter builder is specified at run time (via the `ECFiltering` service configuration option, see 26.5.2.5), the event channel will construct groups with no children and potentially create a filter with different semantics than those intended.
The prefix filter builder supports arbitrarily complex nesting, but requires the arguments to the start group functions that specify the number of children. Always specify the number of immediate children and not the children contained in enclosed groups. Here is an example of a more complex filter with the groups indented to show the intended structure.

```c
ACE_ConsumerQOS_Factory qos;
qos.start_logical_and_group (2);
    qos.start_disjunction_group (2);
    qos.insert_source (MY_SOURCE_1, 0);
    qos.insert_source (MY_SOURCE_2, 0);
    qos.start_disjunction_group (2);
    qos.insert_type (MY_EVENT_1, 0);
    qos.insert_type (MY_EVENT_2, 0);
qos.start_logical_and_group (2);
    qos.start_negation ();
    qos.insert_source (MY_SOURCE_3, 0);
    qos.start_disjunction_group (2);
    qos.insert_type (MY_EVENT_3, 0);
    qos.insert_type (MY_EVENT_4, 0);
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

**Note** Nesting disjunction groups does not make sense for any of the basic filter types discussed so far as it essentially creates one large disjunction group. The only time this structure is needed is when deadline timeouts are being specified. An example of this is shown in 26.3.5.2.

### 26.3.4.11 Quick Rejection Filters Using Bit Masks
A bit-mask filter can be used to quickly reject certain events via bitwise AND operations with source and event type bit masks. Events that have non-zero values for the source and event type after applying the bit masks are passed to the enclosed filter. Events that result in a value of zero for either the source or event type are rejected. Bit-mask filters can only directly contain a single filter (although that filter may be a group that contains other filters). Here is an example of a consumer that uses a bit-mask filter.

```c
ACE_ConsumerQOS_Factory qos;
qos.start_bitmask (0x0000000F, // Source mask
0x0000000F); // Type mask
qos.start_disjunction (2);
qos.insert_source (MY_SOURCE, // Source ID
0); // handle to the rt_info
```
Real-Time Event Service

```cpp
gos.insert_source (MY_SOURCE_2, // Source ID
    0); // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
    gos.get_ConsumerQOS ());
```

This consumer quickly filters out events that have none of the lower four bits set for the source or event type fields. Events with some of these bits set are passed to the disjunction filter. The end result is that all events from MY_SOURCE and MY_SOURCE_2 with an event type that has any one of the lowest four bits set are delivered.

### 26.3.4.12 Null Filters

A null filter will match any event. This type of filter is useful in combination with the bit-mask filter when no further action beyond the bit-mask operation is desired. For example, the following filter group would simply accept all events that have at least one of the four lower order bits set on the source and event type:

```cpp
ACE_ConsumerQOS_Factory gos;
gos.start_bitmask (0x0000000F, // Source mask
    0x0000000F); // Type mask
gos.insert_null_terminator ();
supplier->connect_push_consumer (consumer.in (),
    gos.get_ConsumerQOS ());
```

This consumer receives all events that pass the bit-mask filter.

### 26.3.4.13 Constructing Filters By Hand

For most applications, consumer filters will be constructed using ACE_ConsumerQOS_Factory. However, in some cases it may be necessary to create custom filters. One such case would arise in an application that uses the TAO Real-Time Event Service remotely, but the application itself does not use TAO and is therefore unable to link with the TAO libraries. In this case, the ACE_ConsumerQOS_Factory operations would not be available to the application.

In an application that uses the factory to construct a filter, each call to a start or insert member function on the ACE_ConsumerQOS_Factory object causes the addition of a dependency structure to the ConsumerQOS object that the factory manages. This dependency structure is initialized with an RtecEventComm::Event structure that describes the filter being added.
When `get_ConsumerQOS()` is called on the factory, it returns the managed `ConsumerQOS` object.

Construction of custom filters that do not use the factory is shown in the following example:

```c++
// Create the consumer qos object (describes the filter)
RtecEventChannelAdmin::ConsumerQOS qos;
qos.is_gateway = 0; // This is not a gateway subscription
qos.dependencies.length (2); // set the length of the sequence

// Specify a disjunction group containing one filter
qos.dependencies[0].event.header.type = ACE_ES_DISJUNCTION_DESIGNATOR;
qos.dependencies[0].event.header.source = 1;
qos.dependencies[0].rt_info = 0;

// Specify a filter that accepts events with type of MY_TYPE_1 from any source
qos.dependencies[1].event.header.type = MY_TYPE_1;
qos.dependencies[1].event.header.source = ACE_ES_EVENT_SOURCE_ANY;
qos.dependencies[1].rt_info = 0;
supplier->connect_push_consumer (consumer.in (), qos);
```

A list of the special event types (e.g., `ACE_ES_DISJUNCTION_DESIGNATOR`) can be found in `$TAO_ROOT/orbsvcs/orbsvcs/Event_Service_Constants.h`. More information about the structures associated with specific filter types can be found in the code for the `ACE_ConsumerQOS_Factory` class in the `$TAO_ROOT/orbsvcs/orbsvcs/Event_Utilities.*` files. Note that some filters (such as bit mask and bit-mask value filters) require more than one dependency structure.

### 26.3.5 Timeouts

The TAO Real-Time Event Service also supports the generation of timeout events based on various consumer/supplier criteria. The basic timeouts supported are *interval* timeouts and *deadline* timeouts. Interval timeouts generate a timeout event every time a fixed duration has elapsed. Deadline timeouts deliver timeout events only when certain other dependent events have not been received in the specified duration.

Timeouts are specified using the same filter mechanism as described in the previous section. This means they can be combined with source/event type filters and conjunction/disjunction groups. Such combinations provide
dependencies between timeouts and source/event type filters that result in rich timeout semantics.

Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/EventServices/RTEC_Filter.

### 26.3.5.1 Interval Timeouts

Calling the `insert_time()` operation on the consumer QoS factory with an event type of `ACE_ES_EVENT_INTERVAL_TIMEOUT` specifies an interval timeout. You must also supply the interval desired in 100-nanosecond units.

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_time(ACE_ES_EVENT_INTERVAL_TIMEOUT, 100000000, // 10^8 * 100 nanoseconds = 10 seconds 0); // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This consumer now receives events of type `ACE_ES_EVENT_INTERVAL_TIMEOUT` every 10 seconds. It receives no other events. If other events are desired as well, then additional filters can be added to the disjunction:

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (2);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_time(ACE_ES_EVENT_INTERVAL_TIMEOUT, 100000000, // 10^8 * 100 nanoseconds = 10 seconds 0); // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This consumer now receives all events of type `MY_EVENT_1` as they are sent by the suppliers as well as interval timeout events every 10 seconds.

Placing an interval timeout in a conjunction group with other events causes the timeout event to be delivered with the other events in the group after they have all occurred. Consider the following example:

```c
ACE_ConsumerQOS_Factory qos;
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_time(ACE_ES_EVENT_INTERVAL_TIMEOUT, 100000000, // 10^8 * 100 nanoseconds = 10 seconds 0); // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```
If the timeout event occurs first, the event channel holds the timeout event until a `MY_EVENT_1` type event is received. If a `MY_EVENT_1` type event occurs first, the event channel holds the `MY_EVENT_1` event until a timeout event is received. Both events are delivered together. After one event is queued, any subsequent events of the same type are discarded.

### 26.3.5.2 Deadline Timeouts

Deadline timeouts placed in a disjunction group give the developer the ability to schedule timeouts that occur when a fixed period of time has expired without any of the other events in the group being published. Deadline events are also created with the `insert_time()` member function, but require an event type of `ACE_ES_EVENT_DEADLINE_TIMEOUT`. The timeout value is still specified in 100-nanosecond units. Here is an example that defines a deadline timeout:

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (3);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_type (MY_EVENT_2, 0);
qos.insert_time(ACE_ES_EVENT_DEADLINE_TIMEOUT,
100000000,   // 10^8 * 100 nanoseconds = 10 seconds
0);          // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS());
```

In this example, an event of type `ACE_ES_EVENT_DEADLINE_TIMEOUT` is received by the consumer whenever 10 seconds have elapsed without an event of either type `MY_EVENT_1` or `MY_EVENT_2` having occurred. Each time an event of one of these types occurs, it is delivered to the consumer and the deadline timer is reset.

Placing a deadline timer in a conjunction group causes deadline timeout events whenever all the other filters in the group have not been satisfied before the deadline arrives. Repeating the previous example with a conjunction results in a consumer that only receives deadline timeout events whenever 10 seconds have elapsed without both `MY_EVENT_1` and `MY_EVENT_2` types of events occurring. Each time a matched pair of events is delivered to the consumer, the timer is reset.
Deadline timeouts are the only type of filter that benefits from the existence of nested disjunction groups. These nested groups allow the specification of separate deadline timeout events for different groups of events. Consider the following filter specification:

```c
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (3);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_type (MY_EVENT_2, 0);
qos.insert_time(ACE_ES_EVENT_DEADLINE_TIMEOUT, 100000000, 0);
qos.start_disjunction_group (3);
qos.insert_type (MY_EVENT_3, 0);
qos.insert_type (MY_EVENT_4, 0);
qos.insert_time(ACE_ES_EVENT_DEADLINE_TIMEOUT, 50000000, 0);
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This example delivers events of all four specified types to the consumer as well as timeout events whenever either 10 seconds elapse without events of either type 1 or 2 occurring or 5 seconds elapse without events of either type 3 or 4 occurring. When this consumer receives a timeout, there is no way for it to distinguish which deadline was exceeded. The only way to differentiate is to split it into two consumers, each with its own deadline timeout.

### 26.3.6 Creating and Configuring Event Channel Servants

The previous examples all used event channels that were in a remote process (the Event_Service server). The following section discusses the creation and management of event channel objects from within an application.

There are two separate mechanisms for configuring event channels, each of which provides a different set of configuration options. When creating your own event channels, you can set their attributes to configure certain behaviors. See 26.3.6.2 for the attributes and how to set them. You can always configure event channels using the service configurator. See 26.5 for the service configurator options and how to set them.

#### 26.3.6.1 Local Event Channel Example

The following code shows the steps necessary to create and initialize a local event channel. First the following header files need to be included:

```c
#include <orbsvcs/Event/EC_Event_Channel.h>
#include <orbsvcs/Event/EC_Default_Factory.h>
```
Next, the event channel’s resource factory needs to be initialized and registered. This must occur before the ORB is initialized and is required to allow the event channel to be configured via the service configurator (see 26.5). Failure to initialize the factory results in any event channel configuration options in the service configurator file being ignored. Here is the initialization call:

```cpp
TAO_EC_Default_Factory::init_svcs();
```

After initializing the ORB, and activating the POA Manager, the application is ready to create the event channel object and register it with the POA.

```cpp
// Get the POA/POA Manager and activate the POA Manager
CORBA::Object_var object = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(object.in());
PortableServer::POAManager_var poa_manager = poa->the_POAManager();
poa_manager->activate();

// Create a local event channel and register it with the RootPOA.
TAO_EC_Event_Channel_Attributes attributes(poa.in(), poa.in());
TAO_EC_Event_Channel* ec_impl = new TAO_EC_Event_Channel(attributes);
ec_impl->activate();
PortableServer::ObjectId_var oid = poa->activate_object(ec_impl);
CORBA::Object_var ec_obj = poa->id_to_reference(oid.in());
RtecEventChannelAdmin::EventChannel_var ec =
    RtecEventChannelAdmin::EventChannel::_narrow(ec_obj.in());
```

The event channel is now ready to use. The only step remaining is to make the event channel available to interested consumers and suppliers. The examples in this section have been using the Naming Service to locate the event channel. The event channel can be bound in the Naming Service as follows:

```cpp
const char* ecname = "MyName";
object = orb->resolve_initial_references("NameService");
CosNaming::NamingContextExt_var root_context =
    CosNaming::NamingContextExt::_narrow(object.in());
CosNaming::Name_var name = root_context->to_name(ecname);
root_context->rebind(name.in(), ec.in());
```

The new event channel is now bound to "MyName" in the root naming context of the Naming Service. Other techniques (that are not shown here) of making the event channel implementation available to consumers and suppliers include:
• Returning the event channel object reference through an operation in the application’s IDL interface.
• Writing the event channel’s IOR as a string to a file.
• Using persistent object references.

### 26.3.6.2 Setting Attributes of the Event Channel

The `TAO_EC_Event_Channel_Attributes` object that is passed to the event channel constructor is used to set several event channel attributes. Table 26-6 provides a summary of the attributes that can be set:

**Table 26-6 Event Channel Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>supplier_poa</td>
<td>PortableServer::POA_ptr</td>
<td>None</td>
<td>POA used by supplier admin and supplier proxies. This is typically the same POA the EC uses.</td>
</tr>
<tr>
<td>consumer_poa</td>
<td>PortableServer::POA_ptr</td>
<td>None</td>
<td>POA used by consumer admin and consumer proxies. This is typically the same POA the EC uses.</td>
</tr>
<tr>
<td>consumer_reconnect</td>
<td>int</td>
<td>0</td>
<td>Enables consumer reconnections when non-zero.</td>
</tr>
<tr>
<td>supplier_reconnect</td>
<td>int</td>
<td>0</td>
<td>Enables supplier reconnections when non-zero.</td>
</tr>
<tr>
<td>disconnect_callbacks</td>
<td>int</td>
<td>0</td>
<td>If not zero, the event channel sends disconnect callbacks when a disconnect operation is called on a proxy.</td>
</tr>
<tr>
<td>scheduler</td>
<td>RtecScheduler::Scheduler_ptr</td>
<td>nil</td>
<td>Scheduler that the EC should collaborate with. The default value causes the EC to not use a scheduler.</td>
</tr>
</tbody>
</table>

In the example in the previous section, the only attributes set were the supplier and consumer POAs:

```c
TAO_EC_Event_Channel_Attributes attributes (poa.in (), // Supplier POA
    poa.in ()); // Consumer POA
TAO_EC_Event_Channel* ec_impl = new TAO_EC_Event_Channel(attributes);
```
26.3 Using the TAO Real-Time Event Service

The event channel uses the supplier POA to activate SupplierAdmin and ProxyPushSupplier servants, and the consumer POA to activate ConsumerAdmin and ProxyPushConsumer servants.

All other attributes of the event channel are set using public data members of the TAO_EC_Event_Channel_Attributes class. For example:

```c++
TAO_EC_Event_Channel_Attributes attributes (poa.in (), poa.in ());
attributes.supplier_reconnect = 1;
attributes.consumer_reconnect = 1;
TAO_EC_Event_Channel* ec_impl = new TAO_EC_Event_Channel(attributes);
```

The example above enables suppliers and consumers to reconnect, thereby allowing suppliers to call connect_push_supplier() and consumers to call connect_push_consumer() multiple times without disconnecting as discussed in 26.3.2.3.

The disconnect_callbacks attribute controls whether the consumer and supplier disconnect callbacks are called when the corresponding disconnect operation is called on their proxy object. For example, if this attribute is set to true, when a consumer calls disconnect_push_supplier() on its proxy, the event channel invokes disconnect_push_consumer() on the consumer. Similar behaviors exist for push suppliers.

The scheduler attribute is a reference to the scheduler that the event channel uses. For more details, see the Scheduling Service documentation.

The default values for these attributes are defined as preprocessor macros in $TAO_ROOT/orbsvcs/orbsvcs/Event/EC_Defaults.h. You can use your own project-specific defaults by setting these macros in your config.h file and recompiling the event service.

26.3.7 Observers

The TAO Real-Time Event Channel architecture defines an Observer interface that allows for user-defined CORBA objects that are notified each time suppliers and consumers are connected to or disconnected from an event channel. This functionality is useful for monitoring subscriptions to aid in efficiently federating networks of event channels as discussed in 26.3.8. By default, the observer functionality is disabled and must be enabled using the -ECObserver service configurator option described in 26.5.2.6.
To implement custom gateways or other advanced features, you may need to develop your own observers. To create an observer, use the description in $TAO_ROOT/orbsvcs/orbsvcs/RtecEventChannelAdmin.idl to implement the Observer interface. To connect the observer, use the append_observer() operation in the EventChannel interface. While your observer is connected, and a consumer connects or disconnects, the event channel calls update_consumer() with a ConsumerQOS structure that represents the aggregate subscription information for all consumers. Likewise, each time a supplier connects or disconnects, update_supplier() is called with a SupplierQOS structure that holds the aggregate supplier information. To disconnect the observer, use the remove_observer() operation of the EventChannel interface.

Subscriptions with the is_gateway flag set to true do not have their subscription information passed to observers. This flag is set to false by default and can be set to true via the is_gateway data member of the ConsumerQOS structure.

```cpp
ACE_ConsumerQOS_Factory qosFact;
// Setup the qos...
RtecEventChannelAdmin::ConsumerQOS qos = qosFact.get_ConsumerQOS();
qos.is_gateway = 1;
// Connect to the EC using the qos structure...
```

Gateways can be used to federate event channels (see 26.3.8).

### 26.3.8 Federating Event Channels

Using a single event channel in a large distributed system can often lead to unnecessarily long delivery times and unnecessarily high levels of network traffic. This is especially true when many of the consumers and suppliers that are communicating are located on the same machine or possibly even in the same process. Figure 26-2 shows such a system, where each node contains
consumers and suppliers and almost all events must make two trips across the network.

![Figure 26-2 Single Event Channel System Architecture](image)

**Figure 26-2 Single Event Channel System Architecture**

By having event channels on each node in the network (and dividing the events published between them in some logical manner), you can avoid some of these problems. Figure 26-3 shows a hypothetical system with such an architecture. This configuration avoids some of the problems of the single event channel system. However, the architect must now specify the event channels to use for particular events and consumer and supplier processes must now communicate with several distinct event channels. Even with an
optimal distribution of events on event channels, this configuration may produce additional overhead and network traffic.

Federating event channels avoids many of these problems. When separate event channels are joined into a federation using one of the techniques described in the subsequent sections, the resulting federated event channel acts as one logical event channel. This frees you from allocating events to event channels or interacting with more than one event channel. It also avoids performance problems by delivering events locally, when possible. The Real-Time Event Service provides several mechanisms for federating event channels that differ mainly in the protocol used to transmit events between members of the federation. Federation mechanisms exist that send the events via CORBA messages, UDP, and IP multicast.

These mechanisms all utilize the normal consumer and supplier interfaces to receive events from one event channel and forward them to another. The CORBA and multicast mechanisms make use of the event channel’s `Observer` interface that allows them to automatically tailor their consumer subscriptions based on the consumers that are connected to the event channel they are supplying. See 26.3.7 for a full description of the `Observer` interface.
To ensure that events do not infinitely loop between event channels in the federation, the time to live (ttl) field in the event header is used as a counter. Each time an event passes between event channels in a federation, the ttl flag is decremented. If the ttl value reaches zero, the event is no longer passed between event channels. Typically, you will want to set the ttl field to 1 when using federations. Events of local interest may use a value of 0.

26.3.8.1 Using the CORBA Gateway

TAO defines a generic gateway interface in the TAO_EC_Gateway class and provides one implementation of this interface, TAO_EC_Gateway_IIO, that uses CORBA messages as the underlying communication mechanism. This gateway uses the existing consumer and supplier interfaces to connect to a pair of event channels (one local and one remote). It transmits the events between the event channels by receiving them from a remote event channel as a consumer and sending them to the local event channel as a supplier. To use the consumer information from the local event channel for customizing its subscription information for the remote event channel, the gateway acts as an Observer on the local event channel. This ensures all necessary events (and no more) are passed to it for eventual delivery to the local consumers.

Figure 26-4 shows three event channels that have been federated using CORBA gateways.
Consumers and suppliers using the federation only connect to their local event channel and are unaware of the federation. The local event channel handles all events published from a local supplier to a local consumer. If any remote consumers are interested in them (subscribe to them), they are passed through the appropriate gateway to the corresponding remote event channel.

The following code shows how to create and use a CORBA based federation of two event channels. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/EventServices/RTEC_Federated.

First include the appropriate header file:

```c++
#include <orbsvcs/Event/EC_Gateway_IIOP.h>
```

After the local event channel has been created and a reference to the remote event channel obtained, the gateway can be created and initialized:

```c++
RtecEventChannelAdmin::EventChannel_var local_ec;
RtecEventChannelAdmin::EventChannel_var remote_ec;

// Initialize the local event channel and connect to
// the remote event channel...

TAO_EC_Gateway_IIOP gateway;
gateway.init(remote_ec.in(), // Remote EC object reference
             local_ec.in()); // Local EC object reference
```

The gateway object is initialized with the remote and local event channel references.

To activate the gateway, it must be registered with the POA and attached as an observer to the local event channel. The TAO_EC_Gateway_IIOP class implements the RtecEventChannelAdmin::Observer interface that allows it to be passed into the append_observer() operation.

```c++
CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_obj.in());
PortableServer::ObjectId_var gateway_oid = poa->activate_object(&gateway);
```
26.3 Using the TAO Real-Time Event Service

```cpp
CORBA::Object_var gateway_obj = poa->id_to_reference(gateway_oid.in());
RtecEventChannelAdmin::Observer_var obs =
    RtecEventChannelAdmin::Observer::_narrow(gateway_obj.in());
RtecEventChannelAdmin::Observer_Handle local_ec_obs_handle =
    local_ec->append_observer(obs.in());
```

The gateway is now observing the local event channel and each time a consumer is added or removed it receives the full subscription information for all consumers and updates its subscription on the remote event channel. When the remote event channel delivers events to the gateway (acting as a consumer), it publishes them on the local event channel (acting as a supplier).

All suppliers and consumers now interact with the federated event channel by simply using their local event channel.

To fully federate a set of N event channels using CORBA-based gateways requires that N-1 gateways be collocated with each event channel (one for each remote event channel).

The CORBA gateway can be configured using the IIOP Gateway Factory object as described in 26.6.

26.3.8.2 Federating Event Channels with UDP

TAO’s Real-Time Event Service supports two separate APIs for federating event channels using UDP. The first API has been supported by TAO for a number of years and uses a number of low-level classes that allow you to directly control all aspects of the federation. The second API is new to TAO 1.4a and allows a single TAO_ECG_Mcast_Gateway object to encapsulate all of the lower level objects of the first API. While not fully controllable, the gateway object provides a number of configuration options for commonly used variations. The example in this section uses the lower-level API. For more details on the gateway class and its usage, see the class documentation in $TAO_ROOT/orbsvcs/orbsvcs/Event/ECG_Mcast_Gateway.h.

To federate two event channels using the low-level UDP API, the TAO_EGP_UDP_Sender, TAO_EGP_UDP_Receiver, and TAO_ECG_UDP_EH classes are needed. To actually federate the channels, construct sender and receiver objects, attach each to an event channel, and connect the sender to the remote receiver. Figure 26-5 shows three event channels that have been federated using UDP. The sender object consumes events from its collocated event channel and transmits each event to a specified UDP port (associated with a remote event channel). The event handler (UDP EH) listens on a UDP
port and passes each incoming event to the receiver. The receiver acts as an event supplier and pushes each event received onto its collocated event channel.

The following code shows how an individual event channel can be federated with another similarly configured event channel. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/EventServices/RTEC_MCast_Federated.

First, include the files where the UDP federation classes are defined:

```c
#include <orbsvcs/Event/ECG_UDP_Sender.h>
#include <orbsvcs/Event/ECG_UDP_Receiver.h>
#include <orbsvcs/Event/ECG_UDP_Out_Endpoint.h>
#include <orbsvcs/Event/ECG_UDP_EH.h>
```

After the event channel is created and activated, we need to create and activate an object that implements the `RtecUDPAdmin::AddrServer` interface. This object will be used by the sender object to determine where to send each event that the sender receives. The simple implementation used here

![Figure 26-5 UDP-Based Event Channel Federation](image-url)
26.3 Using the TAO Real-Time Event Service

(SimpleAddressServer) always returns the single address that it was passed in its constructor. That address should refer to the node and port where a corresponding event channel’s event handler is listening.

```c++
    u_short port = 12345;          // Port # of the remote EC’s event handler
    const char* address = "node2"; // The remote EC’s node
    ACE_INET_Addr send_addr (port, address);
    SimpleAddressServer addr_srv_impl (send_addr);
    PortableServer::ObjectId_var addr_srv_oid = 
        poa->activate_object(addr_srv_impl);
    CORBA::Object_var addr_srv_obj = poa->id_to_reference(addr_srv_oid.in());
    RtecUDPAdmin::AddrServer_var addr_srv = 
        RtecUDPAdmin::AddrServer::_narrow(addr_srv_obj.in());
```

We now construct, initialize, and connect the sender object to the event channel as a consumer. We pass the object reference of our simple RtecUDPAdmin::AddrServer to the sender’s init() function so it can use it to get the address to which it should send each event it receives:

```c++
    // Create and initialize the sender object
    TAO_EC_Servant_Var<TAO_ECG_UDP_Sender> sender = TAO_ECG_UDP_Sender::create();
    TAO_ECG_UDP_Out_Endpoint endpoint;
    if (endpoint.dgram ().open (ACE_Addr::sap_any) == -1) {
        std::cerr << "Cannot open send endpoint" << std::endl;
        return 1;
    }
    // Clone endpoint so sender can take ownership
    sender->init (ec.in (), addr_srv.in (),
        new TAO_ECG_UDP_Out_Endpoint (endpoint));
```

Next, we construct, initialize, and connect the receiver object to the event channel as a supplier. Again, we pass the object reference to our simple RtecUDPAdmin::AddrServer to the receiver’s init() function:

```c++
    // Create and initialize the receiver
    TAO_EC_Servant_Var<TAO_ECG_UDP_Receiver> receiver =
        TAO_ECG_UDP_Receiver::create();
    receiver->init (ec.in (),
        new TAO_ECG_UDP_Out_Endpoint(endpoint),
```
// Setup the registration and connect to the event channel
ACE_SupplierQOS_Factory supp_qos_fact;
    supp_qos_fact.insert (MY_SOURCE_ID, MY_EVENT_TYPE, 0, 1);
RtecEventChannelAdmin::SupplierQOS pub = supp_qos_fact.get_SupplierQOS ();
    receiver->open (pub);

We then create and connect the UDP event handler to the reactor, and initialize it with the address where it should listen.

// Create the appropriate event handler and register it with the reactor
TAO_ECG_UDP_EH* udp_eh = new TAO_ECG_UDP_EH (receiver.in());
            udp_eh->reactor (orb->orb_core ()->reactor ());
    u_short listenport = 12345;  // Listen port for this EC’s event handler
    ACE_INET_Addr local_addr (listenport);
    if (udp_eh->open (local_addr) == -1) {
            std::cerr << "Cannot open EH" << std::endl;
    }

Once we enter the event loop and the other event channel is similarly configured, this event channel then forms a federation with the other event channel. Consumers connected to either event channel can receive events from both. Each event pushed to an event channel is automatically sent to the other event channel (assuming its ttl field is one).

One way to connect a third event channel to this federation is to add a second sender object (to each existing event channel) and force it to use the address of the new event channel. In this type of configuration, each event channel in a federation of N event channels requires a receiver and N-1 senders.

For the sake of completeness, here is the implementation of the SimpleAddressServer. First, the header file:

#include <orbsvcs/RtecUDPAdminS.h>
class SimpleAddressServer : public POA_RtecUDPAdmin::AddrServer {
public:
    SimpleAddressServer (const ACE_INET_Addr& address);
    virtual void get_addr (const RtecEventComm::EventHeader& header,
                    RtecUDPAdmin::UDP.Addr& address)
    throw (CORBA::SystemException);

private:
    RtecUDPAdmin::UDP.Addr address_;
26.3 Using the TAO Real-Time Event Service

Next, the implementation of the constructor and the `get_addr()` operation:

```cpp
#include "SimpleAddressServer.h"
#include <ace/INET_Addr.h>

SimpleAddressServer::SimpleAddressServer (const ACE_INET_Addr& address)
{
    this->address_.ipaddr = address.get_ip_address ();
    this->address_.port   = address.get_port_number ();
}

void SimpleAddressServer::get_addr (const RtecEventComm::EventHeader&,
                                      RtecUDPAdmin::UDP_Addr& address)
    throw (CORBA::SystemException)
{
    address = this->address_; 
}
```

26.3.8.3 Federating Event Channels with IP Multicast

The UDP and multicast federation mechanisms share much of the same infrastructure. Like UDP federations, multicast federations can be constructed using either a `TAO_ECG_Mcast_Gateway` object or a set of lower level objects that give developers direct control of the federation. Our example uses the lower level objects of the sender, receiver, and event handler and is very similar to that in the preceding section. For more information about using the gateway object, see the class documentation in `$TAO_ROOT/orbsvcs/orbsvcs/Event/ECG_Mcast_Gateway.h`.

Figure 26-6 shows a multicast federation of three event channels. The main differences from the UDP example, lie in the address supplied to and returned
by the address server (that should now be a multicast address) and the type of
the event handler.

The example discussed in the previous section and found in the
$TAO_ROOT/DevGuideExamples/EventServices/RTEC_Mcast_Federated
directory can also be used to form a multicast federation. The main code
difference is found in the creation and initialization of the event handler. Note
that the multicast event handler is no longer initialized with an address. It
receives the address by querying the address server (via the receiver passed
into its constructor).

```
TAO_ECG_Mcast_EH* mcast_eh = new TAO_ECG_Mcast_EH (receiver.in());
mcast_eh->reactor (orb->orb_core ()->reactor ());
if (mcast_eh->open (ec.in()) == -1) {
  std::cerr << "Cannot open EH" << std::endl;
}
eh = mcast_eh;
```

Because of the nature of multicast, you can use this example to federate an
unlimited number of event channels (as long as they all use the same
port/address combination). This makes the multicast approach the most
scalable mechanism for federating large numbers of event channels.

**Figure 26-6 Multicast-Based Event Channel Federation**

The example discussed in the previous section and found in the
$TAO_ROOT/DevGuideExamples/EventServices/RTEC_Mcast_Federated
directory can also be used to form a multicast federation. The main code
difference is found in the creation and initialization of the event handler. Note
that the multicast event handler is no longer initialized with an address. It
receives the address by querying the address server (via the receiver passed
into its constructor).

```
TAO_ECG_Mcast_EH* mcast_eh = new TAO_ECG_Mcast_EH (receiver.in());
mcast_eh->reactor (orb->orb_core ()->reactor ());
if (mcast_eh->open (ec.in()) == -1) {
  std::cerr << "Cannot open EH" << std::endl;
}
eh = mcast_eh;
```

Because of the nature of multicast, you can use this example to federate an
unlimited number of event channels (as long as they all use the same
port/address combination). This makes the multicast approach the most
scalable mechanism for federating large numbers of event channels.
26.4 Event_Service Command Line Options

The Event_Service server supplies the capability to start a single real-time event channel in its own process. You can configure it to make use of a local (in-process) or global scheduler. This server uses a local scheduler by default. It binds the Event Channel to a supplied name in the root naming context of

---

**Note** Because the TAO_ECG Mcast_EH uses the Observer capabilities of the EC, the Observer functionality must be activated as discussed in 26.5.2.6.

---

26.3.8.4 Choosing The Appropriate Federation Mechanism

Choosing the best federation mechanism for a project involves understanding the relative strengths (and weaknesses) of the existing mechanisms and evaluating them with respect to a project’s unique characteristics and forces.

The multicast federation approach has several unique advantages that make it applicable to many projects. It is the easiest federation type to set up and is also the most scalable in terms of memory usage and network traffic. Its main disadvantages are its use of an inherently unreliable protocol (that is often not supported on some platforms and networks) and potential problems with federating ECs over a complex network. It is appropriate for projects that operate over a controlled network (that would make it fairly reliable) and need to conserve network bandwidth.

The UDP federation is more complex than the multicast federation, and thus does not scale as well. It requires more sender objects than the multicast mechanism for a full federation and thus will use more memory. It may also result in redundant network traffic. Its use of an inherently unreliable protocol is also a disadvantage. The UDP approach is appropriate for systems that cannot use multicast for some of the reasons cited above.

The CORBA federation uses a reliable protocol (by default it uses IIOP, which uses TCP/IP) but forces the addition of many gateway objects as the federation grows, like the UDP mechanism forces the addition of many senders. Thus, it is about as complex as the UDP mechanism. It may result in more network traffic than the UDP mechanism because of its use of a heavier weight transport protocol. It is especially appropriate for relatively small systems that require reliable event delivery.
the Naming Service. The Naming Service server must be running to use this server. The event channel created implements the RtecEventChannel ADMIN::EventChannel interface. Table 26-7 describes the available command line options.

**Table 26-7 Event Service Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-n RTEC_name</td>
<td>Specifies the name with which to bind the Event Channel in the root naming context of the Naming Service.</td>
<td>EventService</td>
</tr>
<tr>
<td>-s (global</td>
<td>local</td>
<td>none)</td>
</tr>
<tr>
<td>-t (new</td>
<td>old_reactive</td>
<td>old_mt)</td>
</tr>
<tr>
<td>-o filename</td>
<td>Specify the name of the file used for storing the IOR of the event channel.</td>
<td>Do not store the IOR in a file.</td>
</tr>
<tr>
<td>-p pid_filename</td>
<td>Specify the name of the file used for storing the process ID of this server.</td>
<td>Do not store the PID in a file.</td>
</tr>
<tr>
<td>-q object_id</td>
<td>Specify an Object ID to use for activating the event channel. When specified, this option also makes the event channel a persistent CORBA object.</td>
<td>Event channel is a transient CORBA object.</td>
</tr>
<tr>
<td>-b</td>
<td>Specifies that this event channel should enable bi-directional GIOP support.</td>
<td>Bi-directional GIOP support is not enabled.</td>
</tr>
</tbody>
</table>

If a local scheduler is specified, the scheduler object is registered in the Naming Service under the root naming context with the name "ScheduleService." If a global scheduler is requested, the process that contains the scheduler should be started before running the event service. For further information see the appropriate Scheduling Service documentation.

Dispatch models for the new event channel implementation are controlled via service configurator options as discussed in 26.5.1.1.
26.5 Event Channel Resource Factory

The event channel resource factory is responsible for creating many strategy objects that control the behavior of the event channel. The behavior of the event channel is typically controlled by using the service configurator file to select the appropriate behaviors for the default factory implementation. In addition to the default factory implementation the RTES supplies a number of other implementation of this factory. Some are used to enable specific features that most applications do not use (such as real-time scheduling and dispatching) and others are minimal implementations that constrain run-time configuration. Optionally, the application developer could also provide a custom implementation of the event channel resource factory containing tailored or customized strategies. This section focuses on the options supported by the default event channel factory and some commonly used alternatives.

The default event channel resource factory is registered with the service configurator using the name EC_Factory. The default event channel resource factory is statically registered with the service configurator, so the static directive is used to supply initialization options to it. To change the behavior of the default event channel factory, add a line similar to the line shown below to your service configuration file.

```
static EC_Factory "-ECFiltering basic -ECDispatching mt"
```

For these options to be effective, you must make sure that the following function call occurs before the ORB is initialized:

```
TAO_EC_Default_Factory::init_svcs ()
```

This creates a default event channel resource factory and statically registers it. If you forget to do this, the service configurator is not able to find the EC_Factory to initialize it.

The -ORBSvcConf option allows you to use file names other than svc.conf for service configurator initialization. See 19.8.32 for more information on this option.

Default values for many of the default event channel resource factory’s options are defined as preprocessor macros in $TAO_ROOT/orbsvcs/orbsvcs/Event/EC_Defaults.h. You can use

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
your own project-specific defaults by setting these macros in your `config.h` file and recompiling the event service.

### 26.5.1 EC_Factory Option Overview

This section provides an overview of the configuration options supported by the default EC_Factory. The following section provides detailed documentation of each of the individual options.

#### 26.5.1.1 Dispatching

When the event channel is pushing events to interested consumers, the thread used to push the event is one decision that has far-reaching effects on the performance and behavior of the application. The event channel resource factory allows for selection of a dispatching strategy that defines how to push events received from suppliers to the interested consumers on the appropriate thread. The default event channel resource factory allows for reactive, priority, and multithreaded dispatching strategies. In addition, when a multithreaded dispatching strategy is selected, the number of threads to be used can be specified. Table 26-8 shows the options related to dispatching strategies.

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-ECDispatching</code></td>
<td>26.5.2.3</td>
<td>Supply this option to select the dispatching strategy for supplier produced events.</td>
</tr>
<tr>
<td><code>-ECDispatchingThreads</code></td>
<td>26.5.2.4</td>
<td>Specify the number of threads to create and use for the multithreaded dispatching strategy. Defaults to one thread.</td>
</tr>
<tr>
<td><code>-ECTimeout</code></td>
<td>26.5.2.15</td>
<td>Supply this option to select the dispatching strategy for timeout events.</td>
</tr>
</tbody>
</table>

The reactive dispatching strategy delivers events on the same thread as they were received (or generated). This is usually the reactor’s main thread. The multithreaded dispatching strategy creates a pool of threads and distributes each event on a randomly-selected member of the pool. The timeout strategy is used to determine how timeout events are dispatched. This option takes only a single value here but is extended by the non-default factories to provide additional strategies.
26.5.1.2 Feature Control

Various features of the event channel can be disabled, configured, and replaced via the event channel resource factory. These include supplier and consumer filtering, observers, ORB identity, and scheduler interaction. Table 26-9 shows the options related to control of these event channel features.

Table 26-9 Event channel feature control options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ECFiltering (null</td>
<td>basic</td>
<td>prefix)</td>
</tr>
<tr>
<td>-ECSupplierFiltering (null</td>
<td>per-supplier)</td>
<td>26.5.2.14 Controls the supplier-side filtering in the event channel. The null option disables filtering and sends all events to a global collection of consumers. Per-supplier filtering maintains separate consumer lists for each supplier.</td>
</tr>
<tr>
<td>-ECSupplierFilter</td>
<td>26.5.2.14</td>
<td>Same as -ECSupplierFiltering.</td>
</tr>
<tr>
<td>-ECObserver (null</td>
<td>basic</td>
<td>reactive)</td>
</tr>
<tr>
<td>-ECScheduling (null</td>
<td>group)</td>
<td>26.5.2.11 Effects coordination between the event channel and scheduling service to enable building the dependency list of consumers and suppliers.</td>
</tr>
<tr>
<td>-ECUseORBId orbid</td>
<td>26.5.2.16</td>
<td>Specifies the orbid of the ORB that the default factory uses.</td>
</tr>
</tbody>
</table>

The two filtering options can be disabled so as to improve efficiency in cases where they are not being used. To allow consumers to use nested filter groups, the -ECFiltering option must be set to prefix. The observer option needs to be enabled when an event channel process uses gateways or has observers attached to it.

The default factory requires an ORB for a variety of operations. It normally uses the default ORB (with a null string for the ORB identifier). Specify an orbid using the -ECUseORBId option to force the default factory to use a different ORB. Typically, this option is used to ensure that the default factory is using the same ORB as was used to activate the event channel.
26.5.1.3 Locking Options

The locking options allow the event channel resource factory to define the lock type desired for various components in the event channel. The default factory allows specification of the lock type for consumer and supplier proxies using the options shown in Table 26-10.

Table 26-10 Locking options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ECProxyConsumerLock</td>
<td>26.5.2.7</td>
<td>Specifies the lock type for the consumer proxy object.</td>
</tr>
<tr>
<td>{null</td>
<td>thread</td>
<td>recursive}</td>
</tr>
<tr>
<td>-ECProxySupplierLock</td>
<td>26.5.2.10</td>
<td>Specifies the lock type for the supplier proxy object.</td>
</tr>
<tr>
<td>{null</td>
<td>thread</td>
<td>recursive}</td>
</tr>
</tbody>
</table>

These options can be set to null to increase performance if the event channel does not access the given components from multiple threads. The default values ensure that the event channel is thread-safe, but recursive locks may be required to avoid deadlocks in certain complex systems.

26.5.1.4 Consumer and Supplier Control Options

The following group of options allows the event channel resource factory to define how the event channel handles dangling (ill-behaved) suppliers and consumers. Consumers and suppliers that remain connected to the event channel when their CORBA objects are no longer accessible from the event channel process are considered ill-behaved. Such consumers and suppliers result when the consumer or supplier process fails to call disconnect, terminates abnormally, or has its node disconnected from the network. The default factory allows specification and configuration of the control policy for consumer and supplier proxies via the options shown in Table 26-11.

Table 26-11 Consumer and supplier control options

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ECConsumerControl</td>
<td>26.5.2.1</td>
<td>Defines the policy for handling ill-behaved consumers.</td>
</tr>
<tr>
<td>{null</td>
<td>reactive}</td>
<td></td>
</tr>
<tr>
<td>-ECSupplierControl</td>
<td>26.5.2.12</td>
<td>Defines the policy for handling ill-behaved suppliers.</td>
</tr>
<tr>
<td>{null</td>
<td>reactive}</td>
<td></td>
</tr>
<tr>
<td>-ECConsumerControlPeriod</td>
<td>26.5.2.2</td>
<td>Defines the polling period in microseconds for the reactive consumer control policy.</td>
</tr>
<tr>
<td>period</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ECSupplierControlPeriod</td>
<td>26.5.2.13</td>
<td>Defines the polling period in microseconds for the reactive supplier control policy.</td>
</tr>
<tr>
<td>period</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The default control policy of null leaves consumers and suppliers connected to the event channel even if the event channel is unable to access them. This allows consumers and suppliers to continue to be connected even in the face of intermittent communications.

The reactive control policy disconnects a consumer or supplier from the event channel the first time the event channel fails to deliver a request to it. It also periodically polls (by default every 5 seconds) all consumer and supplier proxies to ensure their continued connection. Failure to respond to the polling request results in disconnection of the proxy. If the polling period is set to 0, polling is completely disabled.

### 26.5.1.5 Proxy Collection Options

The proxy collection options define the types of collections used to hold consumer and supplier proxies. The default factory allows specification of the collection type for consumer and supplier proxies via the options shown in Table 26-12.

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ECProxyPushConsumerCollection flags</td>
<td>26.5.2.8</td>
<td>Define the characteristics of the collection used to store proxy consumers in the event channel.</td>
</tr>
<tr>
<td>-ECProxyPushSupplierCollection flags</td>
<td>26.5.2.9</td>
<td>Define the characteristics of the collection used to store proxy suppliers in the event channel.</td>
</tr>
</tbody>
</table>

The flags passed to these collection options fall into three separate groups, with each group specifying a different characteristic of the collection. A colon is used as a separator between flags (e.g., `mt:rb_tree:immediate`).

First, the lock type used to control access to the collection can be specified. The `st` flag allows specification of a null lock. The `mt` flag specifies a thread-safe lock. A thread-safe lock is specified by default.

The second characteristic is the actual collection type used. The `list` flag specifies that an ordered list collection is used. The `rb_tree` flag specifies that a collection using a red-black tree is used. The event channel uses an ordered list collection by default.

The third characteristic specifies how concurrent use of the collection is controlled, specifically the case where the collection is being iterated over...
Real-Time Event Service

while a client is attempting an operation that adds or removes a member of the collection. An example is concurrent distribution of events to push consumers (via iteration over the proxy push suppliers collection) while one of the consumers is attempting to disconnect itself. The default factory specifies four different strategies for this characteristic: immediate, copy_on_read, copy_on_write, and delayed.

The immediate flag causes each operation to block until it receives access to the collection. In the example above, the consumer attempting to disconnect blocks until the event distribution iteration completes. Note, that it is possible that the disconnect request may be processed in the same thread as the event distribution (via a nested upcall). If this occurs, immediate access is granted (the thread already has the lock for the collection) and the iterator may be invalidated. It is the developer’s responsibility to ensure that the iterator is not invalidated. Using the -ECDispatching option (see 26.5.2.3) to establish a separate dispatching thread is the most common method for ensuring this validity. (In other words, the immediate collection update flag should not be used with -ECDispatching reactive.)

The copy_on_read flag causes the iterators to copy the collection before proceeding. This allows iterators to release the lock after the copy is made. Subsequent changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is that of the extra performance overhead incurred when the copy of the collection is allocated and replicated.

The copy_on_write flag causes any modifiers to the collection to make copies of the collection before proceeding. This means changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is that of the extra performance overhead incurred when the copy of the collection is allocated and replicated. Note that the copy_on_write strategy makes a copy each time the collection is changed (connect, disconnect, reconnect, or shutdown) and the copy_on_read strategy makes copies each time the collection is iterated.

The delayed flag causes changes to the collection to be queued while iterations are ongoing. When all iterations are completed, the queued modifications are made. The event channel attributes of busy_hwm and
max_write_delay allow bounds to be set on how many iterators access the collection at a time and how many iterators may access it before modification occurs. See 26.3.6.2 for details of these attributes and how to set them.

26.5.2 Event Channel Resource Factory Options
The remainder of this section describes the individual options interpreted by the default event channel factory. These options are applied to the default event channel resource factory by the service configurator as described in 26.5. In addition, we briefly point out which options are affected by the use of the two scheduling factories contained in EC_Kokyu_Factory.h and EC_Sched_Factory.h. See these factory implementations for further details on their use.
26.5.2.1 ECConsumerControl control_policy

Values for control_policy

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Do not discard dangling consumers.</td>
</tr>
<tr>
<td>reactive</td>
<td>Use a reactive policy to discard dangling consumers.</td>
</tr>
</tbody>
</table>

Description
This option specifies the policy used when dealing with dangling consumers. The null control policy never disconnects ill-behaved consumers. The reactive policy disconnects consumers upon the first communication failure.

Usage
Use the reactive control policy when consumers could possibly be destroyed without disconnecting. Use the default control policy, null, when all consumers are guaranteed to disconnect properly (or not at all) and you do not want to incur the overhead of the reactive policy.

Impact
A null consumer control strategy causes degraded throughput when consumers are destroyed without first disconnecting. The reactive strategy requires slightly more overhead in normal operation, may result in consumers having to reconnect when the network quality is degraded, and requires slightly more memory.

See Also
26.5.2.2, 26.5.2.12

Example
static EC_Factory "-ECConsumerControl reactive"
26.5.2.2  **ECConsumerControlPeriod** *period*

**Description**  Sets the period (in microseconds) that the reactive consumer control policy uses to poll the state of the consumers. The default period is 5000000 (5 seconds).

**Usage**  For event channels using the reactive consumer control policy, use this option to control the time to wait between attempted *pings* on each consumer. The reactive consumer control strategy object pings the consumer by invoking CORBA::Object::non_existent() on the consumer's object reference; this is a synchronous call. The -ECConsumerControlPeriod option is ignored when the consumer control policy is not reactive.

**Impact**  Shorter periods require more bandwidth and processing to validate the existence of the consumers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

**See Also**  26.5.2.1, 26.5.2.13

**Example**  `static EC_Factory "-ECConsumerControl reactive -ECConsumerControlPeriod 1000000"`
26.5.2.3 ECDispatching `dispatching_strategy`

Values for `dispatching_strategy`

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive (default)</td>
<td>The reactive strategy delivers events to consumers on the same thread that received them.</td>
</tr>
<tr>
<td>priority</td>
<td>The priority strategy launches a thread for each priority and dispatches each event on the thread corresponding to its priority. This value is only allowed when the TAO_EC_Sched_Factory is used.</td>
</tr>
<tr>
<td>kokyu</td>
<td>The priority strategy launches a thread for each priority and dispatches each event on the thread corresponding to its priority. This value is only allowed when the TAO_EC_Kokyu_Factory is used.</td>
</tr>
<tr>
<td>mt</td>
<td>The mt (multithreaded) strategy delivers events on separate threads from the one that received them, by randomly selecting a thread from a pool. The number of threads in the pool defaults to one, but can be controlled via the <code>-ECDispatchingThreads</code> option.</td>
</tr>
</tbody>
</table>

**Description**

This option controls the dispatching strategy that the event channel uses. The default strategy is reactive.

**Usage**

This strategy determines the order in which the event channel delivers events as well as its overall throughput. The multithreaded strategy should allow for greater performance than the reactive model. The reactive strategy is appropriate when it is acceptable and/or desired for all events to be delivered in the order they are received. The multithreaded strategy is especially effective at reducing the maximum event delivery latency and decoupling the suppliers from the consumers’ execution time, especially in the collocated case.

The priority and kokyu strategies help prevent priority inversions in the event channel by delivering high priority events first. Each require the loading of their respective libraries and factories. Using either with the default factory results in a warning message. They differ only in the scheduler that they use. The kokyu value can also be followed by an optional policy (`sched_fifo` or `sched_rr`) and scope (system or thread).

**Impact**

The reactive model is inappropriate when slow processing by individual consumers can affect other users, greater throughput is desired, or the system requires strict prioritization of events. Note that the reactive strategy delivers events on the same thread where they are received. The ORB’s configuration determines the receiving thread. The multithreaded strategy may increase the
time required to dispatch an event in lightly loaded event channels and also require the allocation of additional resources.

See Also 21.3.5, 26.5.2.4

Examples

static EC_Factory "-ECDispatching mt"

static EC_Factory "-ECdispatching kokyu SCHED_OTHER -ECScheduling kokyu"
26.5.2.4  ECDDispatchingThreads *nthreads*

**Description**  By default, the multithreaded dispatching strategy creates one thread to use for the delivery of supplier-originated events to consumers. Use this option to specify a different number of threads to be created and used.

**Usage**  Using the multithreaded dispatching strategy with the default of one thread provides the user the benefit of separating the dispatching thread from the receiving thread. This allows for a greater decoupling between suppliers and consumers. Use this option to specify additional dispatching threads which results in additional decoupling between consumers as well as potentially increased throughput.

**Impact**  Specifying additional dispatching threads consumes additional resources.

**See Also**  26.5.2.3

**Example**  `static EC_Factory "-ECDispatching mt -ECDispatchingThreads 5"`
26.5.2.5 ECFiltering filter_strategy

Values for filter_strategy

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
<th>Usage</th>
<th>Impact</th>
<th>See Also</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Disables consumer-specified filtering. Each consumer is given a null filter that passes all events.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>basic (default)</td>
<td>Enables consumer-specified filtering for the event channel. Each consumer installs filters as specified in the subscription. The basic strategy enables all filtering types but limits the nesting of filter groups to two levels.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prefix</td>
<td>Enables consumer-specified filtering for the event channel. Each consumer installs filters as specified in the subscription. The prefix strategy enables all filtering types with no limits on the nesting of filter groups.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>priority</td>
<td>Enables consumer-specified filtering for the event channel and also collaborates with the scheduler to build the dependency graph. This value is only allowed when the TAO_EC_Sched_Factory is used.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kokyu</td>
<td>Enables consumer-specified filtering for the event channel and also collaborates with the kokyu scheduler to build the dependency graph. This value is only allowed when the TAO_EC_Kokyu_Factory is used.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

This option specifies the type of filter builder that the event channel uses to construct the filter tree for each consumer. Specifying different strategies allows the disabling of filtering when it is not desired or the enabling of dependency graph building by the scheduling service.

Usage

Either the basic or prefix filtering strategy is usually the desired strategy. The prefix strategy allows complex nesting of filter groups as discussed in 26.3.4.10, but requires filters to be constructed with additional prefix information. By specifying the null filtering strategy, the event channel can be caused to ignore the filters as specified in the subscriptions. The priority or kokyu filtering strategies should be enabled when the system is being used to collect dependency information. This would most likely be done in conjunction with the -ECScheduling option.

Impact

Using the null strategy may cause consumers to process more events and result in extra network traffic. The priority strategy should only be used when collecting dependency information as it is otherwise superfluous. The basic strategy limits the complexity of the filters allowed and may result in the event channel mis-interpreting certain complex consumer filters.

See Also

26.5.2.11

Example

static EC_Factory "-ECFiltering prefix"
26.5.2.6 ECOObserver observer_strategy

Values for observer_strategy

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>(default) Disables the observer feature.</td>
</tr>
<tr>
<td>basic</td>
<td>Enables the observer feature.</td>
</tr>
</tbody>
</table>

Description
Observers are objects that can be connected to the event channel and notified whenever consumers and suppliers are connected and disconnected. This option is used to enable the notification of observers.

Usage
By default, the observer feature is not enabled, which should allow for more efficient operation of the event channel. If the application defines and uses observers, the basic strategy must be used to make them effective. When gateway objects are utilized to federate event channels, then the basic strategy should be used because the gateways use observers internally in their implementation.

Impact
The basic strategy may result in slightly less efficient connection and disconnection. The null strategy causes observers to not be notified and does not allow the use of gateways.

Example
static EC_Factory "-ECObserver basic"
26.5.2.7  ECProxyConsumerLock  *lock_strategy*

**Values for lock_strategy**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Do not use any locking on the proxy consumers.</td>
</tr>
<tr>
<td>thread</td>
<td>Use a thread-safe lock on the proxy consumers. (default)</td>
</tr>
<tr>
<td>recursive</td>
<td>Use a recursive thread-safe lock on the proxy consumers.</td>
</tr>
</tbody>
</table>

**Description**

This option defines the type of lock to use in synchronizing access to the proxy consumer objects.

**Usage**

Single threaded applications can use the null lock to increase the efficiency of the consumer proxy. Multithreaded applications may need to set the lock to recursive in some cases where operations on the proxy consumer may cause recursive access to the proxy consumer. In all other situations, the thread lock should be used.

**Impact**

The null lock causes problems in applications that access the proxy from more than one thread. The thread lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The recursive lock is even more expensive than the thread lock, but is required by applications that must recursively access the lock.

**See Also**

26.5.2.10

**Example**

```
static EC_Factory "-ECProxyConsumerLock recursive"
```
26.5.2.8 **ECProxyPushConsumerCollection flags**

**Description**
This switch controls the type of collection the event channel uses to hold consumer proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., `mt:list`). The allowable flags are described in Table 26-13. Only one flag per type should be specified.

**Table 26-13 Collection Type Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mt</code> (default)</td>
<td>Synchronization</td>
<td>Use a thread-safe lock for the collection.</td>
</tr>
<tr>
<td><code>st</code></td>
<td></td>
<td>Use a null lock for the collection.</td>
</tr>
<tr>
<td><code>list</code> (default)</td>
<td>Collection</td>
<td>Implement the collection using an ordered list.</td>
</tr>
<tr>
<td><code>rb_tree</code></td>
<td></td>
<td>Implement the collection using a red-black tree.</td>
</tr>
<tr>
<td><code>copy_on_read</code></td>
<td>Iterator</td>
<td>Before initiating an iteration of the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td><code>copy_on_write</code></td>
<td></td>
<td>Before initiating a modification to the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td><code>delayed</code></td>
<td></td>
<td>Changes that cannot be made immediately are queued for later execution.</td>
</tr>
<tr>
<td><code>immediate</code> (default)</td>
<td>Iterator</td>
<td>Threads block until they can execute a change to the collection.</td>
</tr>
</tbody>
</table>

For a more detailed discussion of the collection types see 26.5.1.5.

**Usage**
Applications that can guarantee that a consumer proxy collection is only accessed from a single thread can specify the `st` flag to improve performance.

Event channels that connect and disconnect suppliers often and wish to optimize these operations (at the expense of iteration speed), should specify the `rb_tree` flag.

Applications that use the `immediate` flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. Specifying a separate dispatching thread is one way to accomplish this. If you wish to minimize priority inversions between publication and supplier connections/disconnections, then use the `delayed` flag. Collections using the `copy_on_read` flag are only applicable for systems with small numbers of consumer proxies that require low latencies for proxy collection modifications. Collections using the `copy_on_write` flag are only applicable for systems with small numbers of consumer proxies that require low latencies for proxy collection iterations.
Impact

The mt flag incurs additional overhead over the st flag during connection/disconnection of suppliers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of consumer proxy collections (during connection or disconnection of suppliers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Copy on write collections incur dynamic allocation and copy costs for each modification to the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

See Also

26.5.2.9

Example

static EC_Factory "-ECProxyPushConsumerCollection mt:delayed"
26.5.2.9 **ECProxyPushSupplierCollection flags**

**Description**
This switch controls the type of collection the event channel uses to hold supplier proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., \texttt{mt:list}). The flags are described in Table 26-14. Only one flag per type should be specified.

**Table 26-14 Collection Type Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mt} (default)</td>
<td>Synchronization</td>
<td>Use a thread-safe lock for the collection.</td>
</tr>
<tr>
<td>\texttt{st}</td>
<td></td>
<td>Use a null lock for the collection.</td>
</tr>
<tr>
<td>\texttt{list} (default)</td>
<td>Collection</td>
<td>Implement the collection using an ordered list.</td>
</tr>
<tr>
<td>\texttt{rb_tree}</td>
<td></td>
<td>Implement the collection using a red-black tree.</td>
</tr>
<tr>
<td>\texttt{copy_on_read}</td>
<td>Iterator</td>
<td>Before initiating an iteration of the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td>\texttt{copy_on_write}</td>
<td></td>
<td>Before initiating a modification to the collection, a copy of the complete collection is performed.</td>
</tr>
<tr>
<td>\texttt{delayed}</td>
<td></td>
<td>Changes that cannot be made immediately are queued for later execution.</td>
</tr>
<tr>
<td>\texttt{immediate} (default)</td>
<td></td>
<td>Threads block until they can execute a change to the collection.</td>
</tr>
</tbody>
</table>

For a more detailed discussion of the collection types see 26.5.1.5.

**Usage**
Applications that can guarantee that a supplier proxy collection is only accessed from a single thread can specify the \texttt{st} flag to improve performance.

Event channels that connect and disconnect consumers often and wish to optimize these operations (at the expense of iteration speed), should specify the \texttt{rb_tree} flag.

Applications that use the \texttt{immediate} flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. Specifying a separate dispatching thread is one way to accomplish this. If you wish to minimize priority inversions between publication and consumer connections/disconnections, then use the \texttt{delayed} flag. Collections using the \texttt{copy_on_read} flag are only applicable for systems with small numbers of supplier proxies that require low latencies for proxy collection modifications. Collections using the \texttt{copy_on_write} flag are only applicable for systems with small numbers of supplier proxies that require low latencies for proxy collection iterations.
Impact  The `mt` flag incurs additional overhead over the `st` flag during connection/disconnection of consumers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of supplier proxy collections (during connection or disconnection of consumers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

See Also  26.5.2.8

Example  `static EC_Factory "-ECProxyPushSupplierCollection mt:delayed"`
26.5.2.10 **ECProxySupplierLock** *lock_strategy*

Values for *lock_strategy*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Do not use any locking on the proxy suppliers.</td>
</tr>
<tr>
<td>thread</td>
<td>Use a thread-safe lock on the proxy suppliers. (default)</td>
</tr>
<tr>
<td>recursive</td>
<td>Use a recursive thread-safe lock on the proxy suppliers.</td>
</tr>
</tbody>
</table>

**Description**
This option defines the type of lock to use in synchronizing access to the proxy supplier objects.

**Usage**
Single threaded applications can use the null lock to increase the efficiency of the push supplier. Multithreaded applications may need to set the lock to recursive in some cases where operations on the proxy supplier may cause recursive access to the proxy supplier. In all other situations, the thread lock should be used.

**Impact**
The null lock causes problems in applications that access the proxy from more than one thread. The thread lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The recursive lock is even more expensive than the thread lock, but is required by applications that must recursively access the lock.

**See Also**
26.5.2.7

**Example**
```
static EC_Factory "-ECProxySupplierLock recursive"
```
26.5.2.11 ECScheduling \textit{scheduling\_strategy}

Values for \textit{scheduling\_strategy}

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{null} (default)</td>
<td>Disables collaboration with the scheduling service for dependency list collection.</td>
</tr>
<tr>
<td>\texttt{group}</td>
<td>Schedule/filter events as a set when evaluating for delivery.</td>
</tr>
<tr>
<td>\texttt{priority}</td>
<td>Enables collaboration with the scheduler for the building of the dependency list. This value is only allowed when the TAO_EC_Sched_Factory is used.</td>
</tr>
<tr>
<td>\texttt{kokyu}</td>
<td>Enables collaboration with the kokyu scheduler for the building of the dependency list. This value is only allowed when the TAO_EC_Kokyu_Factory is used.</td>
</tr>
</tbody>
</table>

Description

This scheduling strategy controls the scheduling of events for delivery and other scheduler-related coordination. The null and group scheduling options specify whether events should be delivered individually or as groups, respectively. When the null scheduler is specified, events are always evaluated and delivered one at a time, regardless of how they are packaged by the supplier. When the group scheduler is used, events are kept in the group they were published in by the supplier and all events from the original set that pass a consumer’s filter are delivered as a set to that consumer.

The priority and kokyu scheduling strategies use their respective scheduling libraries to schedule event delivery. These strategies are also used for the purpose of building the dependency lists for use by these schedulers.

Usage

The priority and kokyu strategies are used when the application is using one of the real-time schedulers.

Impact

The group strategy is more efficient when groups of events are published and delivered together.

Example

\texttt{static EC\_Factory "-ECScheduling group"}
26.5.2.12 ECSupplierControl control_policy

Values for control_policy

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null (default)</td>
<td>Do not discard dangling supplier.</td>
</tr>
<tr>
<td>reactive</td>
<td>Use a reactive policy to discard dangling supplier.</td>
</tr>
</tbody>
</table>

Description

This option specifies the policy used when dealing with dangling suppliers. The null control policy never disconnects ill-behaved suppliers. The reactive policy disconnects suppliers at the first communication failure.

Usage

Use the reactive control policy when suppliers may be destroyed without disconnecting. Use the default control policy (null) when you can guarantee that all suppliers disconnect properly (or not at all) and you do not want to incur the overhead of the reactive policy.

Impact

A null supplier control strategy causes degraded throughput when suppliers are destroyed without first disconnecting. The reactive strategy requires slightly more overhead in normal operation, may result in suppliers inability to send messages to the event channel or having to reconnect when the network quality is bad, and requires slightly more memory.

See Also

26.5.2.1, 26.5.2.13

Example

```
static EC_Factory "-ECSupplierControl reactive"
```
26.5.2.13  ECSupplierControlPeriod *period*

**Description**  Set the period (in microseconds) the reactive supplier control policy uses to poll the state of the suppliers. The default period is 5000000 (5 seconds).

**Usage**  For event channels using the reactive supplier control policy, use this option to control the time to wait between attempted *pings* on each supplier. The reactive supplier control strategy object pings the supplier by invoking `CORBA::Object::_non_existent()` on the supplier’s object reference; this is a synchronous call. The `-ECSupplierControlPeriod` option is ignored when the supplier control policy is not reactive.

**Impact**  Shorter periods require more bandwidth and processing to validate the existence of the suppliers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

**See Also**  26.5.2.2, 26.5.2.12

**Example**  
```
static EC_Factory "-ECSupplierControl reactive -ECSupplierControlPeriod 1000000"
```
26.5.2.14 ECSupplierFiltering supplier_filter_strategy

**Values for supplier_filter_strategy**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>Disables supplier-based filtering. This results in the event channel keeping a global collection of consumers and attempting a push to each of them.</td>
</tr>
<tr>
<td>per-supplier (default)</td>
<td>Enables supplier-based filtering. This results in the event channel maintaining a (smaller) collection of interested consumers for each supplier and only attempting a push to this subset of consumers.</td>
</tr>
</tbody>
</table>

**Description**

This option controls supplier-based filtering and does not affect which consumers receive events. It does affect the internal data structures of the event channel that determine its memory footprint and performance.

**Note**

This option can also be specified as -ECSupplierFilter and accepts the same values.

**Usage**

Generally, the per-supplier strategy produces the best performance. The null strategy allows for less memory usage (one global list of supplier proxies in the event channel instead of separate ones for each supplier) and may perform essentially the same for cases where all consumers receive events from all or nearly all of the suppliers.

**Impact**

The per-supplier strategy consumes (relatively) large amounts of memory for systems with high numbers of both consumers and suppliers, especially for the case where all consumers receive events from all suppliers. It also increases the connection and disconnection times for consumers and suppliers. The null strategy decreases overall performance in cases where large numbers of consumers and suppliers exist, and each consumer is only interested in the events published by a small number of suppliers.

When suppliers don’t properly specify their event publication via the SupplierQOS structure at connection time, the per-supplier strategy may sometimes cause events to not be delivered to all eligible consumers.

**Example**

```
static EC_Factory "-ECSupplierFiltering null"
```
26.5.2.15 ECTimeout timeout_dispatching_strategy

Values for timeout_dispatching_strategy

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive (default)</td>
<td>The reactive strategy delivers the timeout events to consumers on the main reactor thread.</td>
</tr>
<tr>
<td>priority</td>
<td>The priority strategy launches a thread for each priority and dispatches each timeout event on the thread corresponding to its priority. This option is not yet implemented.</td>
</tr>
<tr>
<td>kokyu</td>
<td>The kokyu strategy launches a thread for each priority and dispatches each timeout event on the thread corresponding to its priority. This option is not yet implemented.</td>
</tr>
</tbody>
</table>

Description: This option controls what thread is used to dispatch timeout events to the consumers.

Usage: The reactive strategy is the only one currently implemented in the default factory. The reactive strategy uses one of ORB’s processing threads to push the timeout events. The priority and kokyu strategies provide the ability to ensure that high priority timeouts are delivered first by using their respective scheduling libraries.

Impact: The reactive strategy may cause priority inversions. The priority and kokyu strategies may cause an increase in the time required to deliver a timeout event on lightly loaded event channels. The priority and kokyu strategies may cause the allocation of additional resources.

See Also: 26.5.2.3

Example: static EC_Factory "-ECTimeout reactive"
26.5.2.16 **ECUseORBId orb-id**

**Description** Sets the name of the ORB that the default factory implementation uses. The default factory creates strategy objects that use this ORB to perform remote invocations and to gain access to the ORB’s reactor.

**Usage** This option is only useful in applications that create multiple ORBs and activate the event channel in one of them. Use it to ensure that the objects created by the default factory use the same ORB as the event channel and related objects.

**Impact** This option may cause the creation of a new ORB (and associated resources), if the ORB with the given name has not been initialized.

**Example**
```
static EC_Factory "-ECUseORBId Orb2"
```
26.6 The IIOP Gateway Factory

The IIOP gateway factory is responsible for creating strategy objects that control the behavior of the CORBA gateways used for federating event channels as described in 26.3.8.1. The behavior of the CORBA (or IIOP) gateway is controlled by using the service configurator file to select the appropriate behaviors for the default factory implementation.

This resource factory is registered with the service configurator using the name \texttt{EC\_Gateway\_IIOP\_Factory}. It is statically registered with the service configurator, so the static directive is used to supply initialization options to it. To change the behavior of the IIOP gateway factory, add a line similar to the line shown below to your service configuration file.

\begin{verbatim}
static EC_Gateway_IIOFP_Factory "-ECGIIOPConsumerECControl reactive"
\end{verbatim}

For these options to be effective, you must make sure that the following function call occurs in every process containing CORBA gateways \textit{before} the ORB is initialized:

\begin{verbatim}
TAO\_EC\_Gateway\_IIOP\_Factory::init\_svcs ()
\end{verbatim}

This creates an IIOP gateway factory and statically registers it. If you forget to do this, the service configurator is not able to find the \texttt{EC\_Gateway\_IIOP\_Factory} to initialize it.

The -\texttt{ORBSvcConf} option allows you to use file names other than \texttt{svc.conf} for service configurator initialization. See 19.8.32 for more information on this option.

Default values for many of the IIOP gateway factory’s options are defined as preprocessor macros in \texttt{$TAO\_ROOT/orbsvcs/orbsvcs/Event/ECG\_Defaults.h}. You can use your own project-specific defaults by setting these macros in your \texttt{config.h} file and recompiling the event service.

26.6.1 Option Overview

This section provides an overview of the configuration options supported by the default \texttt{EC\_Gateway\_IIOP\_Factory}. Table 26-15 presents a brief summary of the available options for this factory followed by short
descriptions of these options. The following section provides full
documentation for each option.

**Table 26-15 IIOP Gateway Factory Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECGIIOPConsumerECControl</td>
<td>26.6.2.1</td>
<td>Define the policy for handling consumer event channels that can’t be</td>
</tr>
<tr>
<td>{ null</td>
<td>reactive</td>
<td>reconnect }</td>
</tr>
<tr>
<td>ECGIIOPConsumerECControlPeriod</td>
<td>26.6.2.2</td>
<td>Define the period in microseconds of the reactive and reconnect control</td>
</tr>
<tr>
<td>period</td>
<td></td>
<td>policies.</td>
</tr>
<tr>
<td>ECGIIOPConsumerECControlTimeout</td>
<td>26.6.2.3</td>
<td>Round-trip timeout in microseconds for the consumer event channel ping.</td>
</tr>
<tr>
<td>timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECGIIOPUseConsumerProxyMap</td>
<td>26.6.2.4</td>
<td>Flag that specifies whether a map of consumer proxies should be used or a</td>
</tr>
<tr>
<td>{ 0</td>
<td>1 }</td>
<td></td>
</tr>
<tr>
<td>ECGIIOPUseORBId orbid</td>
<td>26.6.2.5</td>
<td>Specifies the id of the ORB that the factory uses.</td>
</tr>
<tr>
<td>ECGIIOPUseTTL</td>
<td>26.6.2.6</td>
<td>Flag that specifies whether the ttl field of the event is used to limit</td>
</tr>
<tr>
<td>{ 0</td>
<td>1 }</td>
<td></td>
</tr>
</tbody>
</table>

The consumer control policy works similarly to the control policies on the event channel, but in the gateway’s case they determine how the downstream or consumer event channel is treated when the gateway fails to invoke a request on it. The control period and timeout affect the period and round-trip timeout of the periodic pings performed by the reactive and reconnect control policies.

Consumer proxy map and TTL (time to live) features can each be disabled for applications that would benefit from their avoidance. Most applications should keep the defaults. Specification of an alternate ORB ID for the factory is beneficial when the gateway’s process is using multiple ORBs.

**26.6.2 IIOP Gateway Factory Options**

The remainder of this section describes the individual options interpreted by the IIOP Gateway factory. These options are applied to the IIOP Gateway factory by the service configurator as described in 26.6.
### 26.6.2.1 ECGIIOPConsumerECControl control_policy

#### Values for control_policy

<table>
<thead>
<tr>
<th>control_policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null (default)</td>
<td>Do not discard consumer event channels that cannot be contacted.</td>
</tr>
<tr>
<td>reactive</td>
<td>Use a reactive policy to discard consumer event channels that cannot be contacted.</td>
</tr>
<tr>
<td>reconnect</td>
<td>Attempts to periodically reconnect to event channels after they cannot be contacted.</td>
</tr>
</tbody>
</table>

#### Description

This option specifies the policy used by the gateway when dealing with consumer event channels that suffer failures when the gateway pushes events on them. The consumer event channel is the event channel that is connected as a consumer to the gateway. The null control policy never disconnects consumer event channels that suffer communication failures and continues to try to push events to those channels. The reactive policy discards consumer event channels upon their first communication failure and never reconnects them. The reconnect control policy also disconnects consumer event channels after their first failure, but attempts to periodically reconnect to that event channel. Both the reactive and reconnect policies periodically poll consumer event channels to determine if they are still available. See the ECGIIOPConsumerECControlPeriod and ECGIIOPConsumerECControlTimeout options for details of this polling.

#### Usage

Use the reconnect control policy for the greatest robustness when using IIOP gateways. Use the default control policy, null, when you can guarantee the reliability of your event channels and you do not want to incur the overhead of the reactive or reconnect policies.

#### Impact

A null consumer control strategy causes degraded throughput when consumer event channels crash unexpectedly or cannot be contacted. Since the reactive strategy never reconnects consumer channels, it causes permanent breaks in the federation. The reconnect strategy requires slightly more overhead in normal operation, may result in reconnections when the network quality is degraded, and requires slightly more memory. When using the reconnect policy, persistent object references for event channels are required.

#### See Also

26.6.2.2, 26.6.2.3

#### Example

```java
static EC_Gateway_IIOP_Factory "-ECGIIOPConsumerECControl reconnect"
```
### 26.6.2.2 ECGIIOPConsumerECControlPeriod \textit{period}

**Description**
Set the period (in microseconds) the reactive and reconnect control policies use to poll the state of the consumer event channels. The default period is 5000000 (5 seconds).

**Usage**
For gateways using the reactive or reconnect control policies, use this option to specify the time to wait between attempted \textit{pings} on each consumer event channel. The reactive and reconnect control strategy objects ping the consumer event channel by invoking \texttt{CORBA::Object::\_non\_existent()} on the event channel’s object reference; this is a synchronous call. This option is ignored when the control policy is \texttt{null}.

**Impact**
Shorter periods require more bandwidth and processing to validate the existence of the event channels. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

**See Also**
26.6.2.1, 26.6.2.3

**Example**
```
static EC_Gateway_IIOFactory "-ECGIIOPConsumerECControl reconnect
-ECGIIOPConsumerECControlPeriod 1000000"
```
26.6.2.3 **ECGIOPConsumerECControlTimeout** *timeout*

**Description**
Sets the relative round-trip timeout (in microseconds) that the reactive and reconnect control policies use for polling consumer event channels. The default timeout is 10000 (10 milliseconds).

**Usage**
For gateways using the reactive or reconnect control policies, use this option to control the time the gateway waits for a consumer event channel to respond to an attempted *ping*. These control strategies ping the consumer by invoking `CORBA::Object::_non_existent()` on the consumer event channel’s object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the control policy classifying that ping as a communication failure for that consumer. This option is ignored when the consumer control policy is null.

**Impact**
Smaller timeout values may result in more timeout failures and consumer event channels being disconnected more often. A larger timeout value means it takes longer to detect and remove event channels that have crashed or become isolated.

**See Also**
26.6.2.1, 26.6.2.2

**Example**
```bash
static EC_Gateway_IIOP_Factory "-ECGIOPConsumerECControl reconnect
-ECGIOPConsumerECControlTimeout 50000"
```
26.6.2.4 ECGIIOPUseConsumerProxyMap flag

Values for flag

<table>
<thead>
<tr>
<th>flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Use a single consumer proxy for all events the gateway processes.</td>
</tr>
<tr>
<td>1 (default)</td>
<td>Use a consumer proxy map with a separate consumer proxy for each source ID the gateway processes.</td>
</tr>
</tbody>
</table>

Description  How many consumer proxies the gateway uses on the consumer event channel affects the performance of your federated event channel application. Using the consumer proxy map means the gateway uses a separate consumer proxy for each source ID and allows the consumer event channel to use supplier filtering to improve efficiency. It also means that changes in the subscription set of the gateway can be handled in a more incremental manner. Not using the map means only a single consumer proxy is used and it supplies all events to the consumer event channel.

Usage  By default, the consumer proxy map is enabled; pass a value of zero to disable it. Most applications probably want this feature enabled but some applications with memory-usage concerns may benefit from disabling it.

Impact  Using the consumer proxy map consumes additional resources in the gateway, but should be more efficient in most applications. Disabling the consumer proxy map consumers less resources, but usually at a performance cost.

See Also  26.5.2.14

Example  static EC_Gateway_IIOP_Factory "-ECGIIOPUseConsumerProxyMap 0"
26.6.2.5  **ECGIOPUseORBId orbid**

**Description**  Sets the name of the ORB used by the gateway factory. The factory creates strategy objects that use this ORB to perform remote invocations and to gain access to the ORB’s reactor.

**Usage**  This option is only useful in applications that create multiple ORBs and activate the gateways in one of them. Use it to ensure that the objects created by the IIOP gateway factory use the same ORB as the event channel and related objects. When not specified, the default ORB is used.

**Impact**  This option may cause the creation of a new ORB (and associated resources), if the ORB with the given name has not been initialized.

**See Also**  26.5.2.16

**Example**  static EC_Gateway_IIOP_Factory "-ECGIOPUseORBId myorb"
26.6.2.6 ECGIIOPUseTTL flag

Values for flag

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Do not use the ttl field. All events pushed to the gateway are pushed to the consumer event channel.</td>
</tr>
<tr>
<td>1 (default)</td>
<td>The ttl field of the event header structure is used to limit the number of gateways an event can pass through.</td>
</tr>
</tbody>
</table>

Description

This option determines whether the gateway uses the ttl field of the event header structure to limit the number of gateways that an event can pass though. When enabled, as it is by default, the ttl field is decremented each time an event passes through a gateway and if the ttl field is zero the gateway does not pass the event. Passing zero to this option disables this behavior and means all events are passed through all gateways.

Usage

For normal gateway behavior, leave the ttl feature enabled. Disabling it may make things simpler for suppliers because they no longer have to set the ttl field which requires knowledge of the federation topology.

Impact

When disabling the ttl functionality events cycle forever in topologies with recursive loops in the federation. This option has negligible effect on performance.

Example

static EC_Gateway_IIOP_Factory "-ECGIIOPUseTTL 0"
CHAPTER 27

Notification Service

27.1 Introduction

The OMG Notification Service version 1.1 specification (OMG Document formal/04-10-13) extends the OMG Event Service (Chapter 25) with many improvements, including: a well-defined event structure; event filtering; discovery of event types required or offered; configurable quality of service properties; and an optional event type repository. The basic architectural elements of the Event Service (consumers, suppliers, event channels, proxy consumers and suppliers, consumer and supplier administration interfaces) are preserved in the Notification Service specification. In fact, implementations of the Notification Service can still support existing Event Service consumers and suppliers without recompilation. New architectural elements (filters, event channel and filter factories, new structured supplier and consumer interfaces) have been added, and these newer features can be accessed using the familiar programming model introduced in the Event Service.

TAO’s implementation of the OMG Notification Service supports a useful subset of the specification. It supports the push-model of event communication only; there is no support for the pull model. Certain optional
features from the specification are also unsupported, including: event type repository, and typed events. Other differences between the specification and TAO’s implementation will be noted in the relevant sections of this chapter.

27.2 Notification Service Architecture

Figure 27-1 shows the architecture of the Notification Service.

The Notification Service is hierarchical in structure, with a single EventChannelFactory supplying one or more EventChannel objects. Each event channel, in turn, supplies one or more ConsumerAdmin and SupplierAdmin objects, and each administration object supplies zero or more consumers or suppliers. Each of these objects is assigned a unique identifier that can be used to retrieve its object reference. Each “parent” interface provides an operation that may be used to enumerate its “children,” and each child provides an attribute for accessing its parent.

The primary purpose of the administration (admin) interfaces is to allow grouping of consumers and suppliers with common filtering and quality of service (QoS) properties. As consumers and suppliers are created, they inherit the applicable QoS properties of their parent admin object. If you later change the QoS properties of an admin, its children will not be updated. Filtering
works differently. The consumers and suppliers do not inherit the parent’s filters; instead, the admin filters are always combined with its children’s filters using either a logical AND or a logical OR operation. Updating the filtering constraints on an admin object has an immediate affect on the event processing of its children.

Consumers and suppliers are derived from one of four different hierarchies. The original Event Service interfaces can still be used to communicate untyped events as simple CORBA anys. In addition, the Notification Service defines three new sets of interfaces; one for use with untyped events, one for structured events, and one for batches of events.

Regardless of the specific consumer and supplier interfaces used, the same basic model of communication is supported. Suppliers push events to the event channel and the event channel asynchronously pushes these events to connected consumers. Each event may pass through filters before being forwarded to consumers, and QoS properties may affect the processing of events as they travel through the various elements of the Notification Service architecture.

The Notification Service supports mixing and matching of the various consumer and supplier interfaces. For example, you can supply events as sequences of structured events using the SequenceProxyPushSupplier interface, and consume them as individual StructuredEvents using the StructuredProxyPushConsumer interface. There are well-defined translations among each of the event propagation mechanisms.

27.3 Notification Service Features

The Notification Service provides several new features, relative to the Event Service, for event-based communications, including:

- Structured Events
- Event Filtering
- Subscriptions
- Batched Events
- Quality of Service Properties

These features are discussed in the next several subsections.
27.3.1 The Structured Event Type

Like the Event Service, the Notification Service defines a model for communicating untyped events in the form of CORBA anys. Untyped events can be used with the new filtering and QoS features, but it is far easier to use the new `CosNotification::StructuredEvent` type. The structured event type has a well-defined data structure comprising three fixed header fields, a variable-length header portion of name-value pairs, a variable-length filterable body portion also of name-value pairs, and a remaining body portion, or payload, that consists of a single CORBA any.

Figure 27-2 shows the `CosNotification::StructuredEvent` type.

![Figure 27-2 CosNotification::StructuredEvent](image_url)
QoS settings for priority, timeout, and reliability may be specified in the variable header portion. The variable body portion can be easily used in filtering constraints. The StructuredEvent is intended for use with applications that require strongly-typed events. Users of these events are able to map their application-specific events into a common data structure, thereby making various optimizations possible during event processing.

The Notification Service specification defines a CosNotifyComm module that contains StructuredPushSupplier and StructuredPushConsumer interfaces for communicating structured events.

27.3.1.1 The Structured Event Header

The structured event’s header has two parts: a fixed header portion containing fields to identify the event’s type and instance; and an optional list of name-value pairs. The three fixed header fields are all strings:

- domain_name—To identify the vertical industry domain to which this event belongs, such as telecommunications, finance, or health-care.
- type_name—To further classify the event within its domain.
- event_name—To identify a specific event instance.

Together, the domain_name and type_name fields are known as the event type, and have special meaning for filtering and subscription processing.

The optional header portion consists of a sequence of zero or more name-value pairs in which QoS settings can be carried. TAO supports two well-known settings for this portion of the header:

- priority—To specify an integer priority value for the event.
- timeout—To specify a relative time after which the event will be discarded if it has not been delivered.

Other name-value pairs in the header can be used to carry data or application-specific header information, but you will normally use the name-value pairs in the body portion, instead.

27.3.1.2 The Structured Event Body

The data content of an event is mapped into the event body. The structured event body is divided into two parts:
- The filterable body portion is intended to carry the most interesting fields of the event, upon which the consumer is most likely to base filtering decisions. It comprises a sequence of name-value pairs, in which each name is a string and each value is a CORBA any.

- The remaining body area is defined as a single CORBA any and is intended to carry large blocks of data related to the event. Although this field is considered separate from the filterable body area of the event, there are no restrictions against additional filtering on the contents of this data.

### 27.3.2 EventBatch Data Type

The Notification Service specification defines a sequence of StructuredEvents called a CosNotification::EventBatch. Within the CosNotifyComm module are SequencePushSupplier and SequencePushConsumer interfaces for communicating with batched events.

The StructuredEvents in an EventBatch received from a supplier are handled independently inside the Notification Service. There is no functional difference between using a series of CORBA calls to send one StructuredEvent at a time, and using a single CORBA call to send a batch of StructuredEvents. There may, however, be a performance benefit to sending a batch of events due to the reduced number of CORBA calls.

The QoS properties MaximumBatchSize and PacingInterval can be applied to consumers that register to receive event batches. MaximumBatchSize is used to specify how many individual StructuredEvents will be queued, and PacingInterval specifies how long StructuredEvents can be queued, before being delivered as a batch. These and other QoS properties are discussed in detail in 27.3.5.

There is no correlation between incoming batches and outgoing batches. For example: if a supplier pushes five events in a batch, a consumer should not expect to receive these five events in a single batch. They may be split across batch boundaries; intermixed with events from other consumers, etc.

### 27.3.3 Event Filtering

The most notable improvement the Notification Service specification introduces over the Event Service is the introduction of event filtering. Event filtering allows a consumer to subscribe to a precise set of events. TAO
supports the Extended Trader Constraint Language (ETCL) filtering grammar as defined by the OMG. ETCL allows applications to create complex expressions to describe which events should be allowed to pass through an element of the Notification Service architecture. Filters are usually applied to admin objects, but may also be used with ProxyConsumer or ProxySupplier objects.

**Note** The filters described above are known as Forwarding filters, because they are used to determine which events should be forwarded to the next layer of the architecture. The Notification Service specification also defines a type of filters known as Mapping filters, which can be used to modify an event’s properties as it passes. Mapping filters are not supported in TAO’s implementation of the Notification Service.

Besides the Extended Trader Constraint Language (ETCL), TAO’s implementation of the Notification Service also supports the original Trader Constraint Language (TCL) as defined by the Object Trader Service specification. For more information on standard TCL, see *Advanced CORBA Programming with C++*, 19.10. For more information on ETCL, see 2.4.2 in the Notification Service specification.

TAO’s implementation of the Notification Service specification supports filtering for both untyped events and structured events. Filtering with structured events is easier than with untyped events, because you can simply add named fields into the variable body portion of the events and define filtering constraints using ETCL that reference the named fields.

**Note** Filtering of untyped events is nominally supported in TAO, but is not recommended.

### 27.3.3.1 Using Event Filtering

Depending upon the characteristics of your application, you can use filters to improve performance. For example, you could use a filter object with a supplier to ensure that the supplier publishes only those events that match a certain set of constraints, and to avoid populating the notification channel with unnecessary events. Similarly, you could use a filter object with a consumer to inform the notification channel of the types of events the consumer wants to
receive and thereby avoid burdening the consumer with having to process unwanted events. In both cases, filtering can also help prevent unnecessary network bandwidth consumption because unwanted events will not be sent at all. Keep in mind, however, that event filtering imposes a computational burden on the processes in which the notification channel itself is operating, over and above its normal responsibilities of receiving and dispatching events.

Filter objects are used to filter events based on a set of constraints provided by the application developer. Filter objects can be attached to an administration object or to individual proxy objects. When a filter object is attached to an administration object, event receiving and forwarding by all the associated proxies is affected. When a change is made to a filter object that is attached to an administration object, all the associated proxies are affected by the change. If you want to control the filtering behavior of each proxy individually, you can attach a filter to each proxy. In this way, the filter affects only the event receiving and forwarding behavior of that individual proxy.

Note that each proxy can have two sets of filter objects associated with it, those that are associated with its managing administration object and those that are associated with that proxy itself. The InterFilterGroupOperator flag can be used to control whether each proxy created by the administration object will perform a logical AND or a logical OR on the results of the two filter sets in making its event receiving and forwarding decisions. If no filters are attached to an object, the default behavior is to pass all events.

Filters are first-class CORBA objects. Filter objects can be collocated with the notification channel process itself or they can be distributed in their own address spaces. The event channel provides a factory interface to create filter objects, and filters created with this factory reside in the same address space as the notification channel itself. You could also create filter objects separately from the event channel and attach them, by object reference. However, filtering via remote filter objects may introduce a significant performance penalty since each filter’s match() operation would be invoked in a distributed fashion.

27.3.4 Offers and Subscriptions

Offers and subscriptions provide a mechanism to allow consumers to be notified whenever the set of offered event types changes, and for suppliers to be notified whenever the set of event types required by consumers changes. By using these offer and subscription notifications, applications can create
adaptive suppliers and consumers that can change their filtering constraints dynamically to adapt to changes in the types of events actually being used in the system. Unlike filters, which work automatically, it is up to the application developer to make use of offer and subscription information in consumer and supplier implementations.

**Note** The Notification Service specification (section 2.6.5) defines a mechanism that ties filtering to offer- and subscription-change notifications, but this feature is not supported in TAO.

### 27.3.4.1 Offer Changes

A supplier uses the `offer_change()` operation to specify the event types it offers to the notification channel. The information passed via the `offer_change()` operation is used by the notification channel to aggregate a list of all the event types that its connected suppliers offer. For example, if three suppliers all offer event type “A”, then the channel only needs to notify the consumers of a change if all three suppliers stop supporting that event type. If a supplier decides to add a new event type or to remove an existing event type from its offer, it can use the `offer_change()` operation again to inform the notification channel of the change in the types of events it supplies.

The `offer_change()` operation can be invoked on the proxy consumer object or on the supplier administration object. Invoking `offer_change()` on the proxy consumer affects only the offer for that particular supplier. Invoking the operation on the supplier administration object, on the other hand, means the change in event types being offered will be shared by all the proxy consumer objects that were created by that `SupplierAdmin` object.

The notification channel invokes `offer_change()` on the consumer whenever a supplier changes its offered set of events and this causes the channel’s set of offered events to change.

### 27.3.4.2 Subscription Changes

Similarly, a consumer uses the `subscription_change()` operation to inform the notification channel of the event types it is interested in receiving. The notification channel uses this information to aggregate a list of all the event types its connected consumers require. For example, if three consumers all want event type “A”, then the channel only needs to notify the suppliers of
a change if all three consumers stop wanting that event type. If the consumer is later required to receive a new event type, or to stop receiving a particular event type, it can use the `subscription_change()` operation again to inform the notification channel of the change in the types of events it requires.

The `subscription_change()` operation can be invoked on the consumer administration object or on the proxy supplier object. Invoking it on the proxy supplier affects only the subscription for that particular consumer, whereas invoking it on the consumer administration object means the change in event types being subscribed to will be shared by all the proxy supplier objects that were created by that `ConsumerAdmin` object.

The notification channel invokes `subscription_change()` on the supplier whenever a consumer changes its subscribed set of events and this causes the channel’s set of subscribed events to change.

### 27.3.4.3 Obtaining Offered and Subscribed Event Types

A supplier can discover the set of event types the consumers of an event channel require by invoking the `obtain_subscription_types()` operation on its proxy consumer. Similarly, a consumer can discover the set of event types suppliers of the channel offer by invoking the `obtain_offered_types()` operation on its proxy supplier.

Once connected to the notification channel, consumers and suppliers are updated with offer and subscription information via the `offer_change()` and `subscription_change()` operations, respectively.

**Note**  
These updates can be disabled via the `-NoUpdates` service configurator option described in 27.7.1.11.

Using these operations, you can create suppliers that are able to respond to the needs of consumers by producing events the consumers require and to stop producing events that are no longer required by any consumers of the notification channel. Likewise, you can create consumers that are able to dynamically change their subscriptions to start receiving new event types added by suppliers or to discontinue their interest in event types that are no longer produced by the suppliers.
27.3 Notification Service Features

27.3.5 QoS Support

Another major feature of the Notification Service that is not found in the Event Service is the inclusion of interfaces to control QoS characteristics of the event delivery. These interfaces include the ability to get and set QoS properties at the event channel, admin, proxy, and event levels.

Note that the Notification Service style of asynchronous event communication does not provide a one-button QoS setting to solve all QoS-related problems. Since there is no direct communication between suppliers and consumers of events, it is not possible to set QoS at just one place such that it covers the entire event communication pathway. Instead, QoS properties must be set at all three conceptual points through which an event can be transmitted—from the supplier to the event channel, within the event channel itself, and from the event channel to the consumer. These three points must be used together to achieve correct quality of service end-to-end.

In addition, many of the following properties also require application of a proper threading model in order for the QoS properties to have their full effect. See 27.7.1 for a discussion of the threading model and related run-time configuration options.

27.3.5.1 Supported QoS Properties

TAO’s implementation of the Notification Service supports a subset of the QoS properties defined by the specification as well as some TAO-specific properties. The following specification-defined properties are not supported:

- StartTime
- StopTime
- StartTimeSupported
- StopTimeSupported

Note: TAO does not support use of the validate_qos() and validate_event_qos() operations, and it does not always raise the UnsupportedQoS and UnsupportedAdmin exceptions where applicable.

The name of each property defined by the specification is contained in a string constant in $TAO_ROOT/orbsvcs/orbsvcs/CosNotification.idl.
Each property also has a specific IDL type for its value and it’s your responsibility to put the correct type of value in each property’s any. The above file also contains constant values that predefine values for certain properties. TAO-specific property details are similarly defined in $TAO_ROOT/orbsvcs/orbsvcs/NotifyExt.idl. The following sections describe TAO’s supported QoS properties along with the associated IDL data type and any predefined values.

### 27.3.5.2 Notification QoS Properties

The following specification-defined QoS properties are established using the `set_qos()` operation and accessed using the `get_qos()` operation.

**Timeout**

This property is only supported on a per-event basis. If an event has not been delivered by the specified relative time, then it will be discarded. The data type of this property is `TimeBase::TimeT`, which is a typedef to an unsigned long long with units of 100s of nanoseconds (10E-7 seconds).

**Priority**

This property is only supported on a per-event basis, and is used to control the order of the events delivered to the consumers. The default value is 0; any integer in the range -32767 to 32767 is valid. The data type is a short. Constants exist for `LowestPriority`, `HighestPriority`, and `DefaultPriority`.

**OrderPolicy**

This property is used by a proxy to arrange the events in its dispatch queue. When events are delivered as published, and not queued, it has no effect. This property only applies to structured events, either individually or within sequences. The following constants define the only valid values which can be assigned to the `OrderPolicy` property.

- **AnyOrder**: According to the specification, events can be delivered in any implementation-specific order. In TAO’s implementation this is equivalent to `FifoOrder`. This is the default value for this policy.
- **FifoOrder**: Events are delivered in the order of their arrival.
- **PriorityOrder**: Events are ordered based on their priority, the highest priority events being delivered first.
• **DeadlineOrder**: Events are ordered based on their expiration timeouts, the events with the shortest timeouts being delivered first.

**DiscardPolicy**
This property defines the order in which the events are discarded by a proxy or the event channel when their internal buffers overflow. This property applies on a per-channel basis only if it is set on a channel that also has the `RejectNewEvents` admin property set to false. The following constants define the only valid values which can be assigned to the `DiscardPolicy` property.

• **AnyOrder**: According to the specification, events can be discarded in any implementation-specific order. In TAO’s implementation this is equivalent to `FifoOrder`. This is the default value for this policy.
• **FifoOrder**: Events are discarded in the order of their arrival.
• **LifoOrder**: Events are discarded in the inverted order of their arrival.
• **PriorityOrder**: Events are discarded based on their priority, the lowest priority events being discarded first.
• **DeadlineOrder**: Events are discarded based on their expiration times, the events with the earliest deadline being discarded first.

**MaxEventsPerConsumer**
This property defines a bound to the maximum number of events the channel will queue on behalf of a given consumer. This property can only be set on proxy suppliers and has a default value of zero, which means that no maximum will be enforced. The data type for this property is long.

**MaximumBatchSize**
This property defines the maximum number of events that will be delivered within each sequence of events. It applies only to sequence proxy suppliers, and defaults to zero, which means that no maximum will be enforced. The data type for this property is long.

**PacingInterval**
This property defines the maximum period of time the channel will collect individual events into a sequence before delivering the sequence to the consumer. The time starts when a new event arrives for an idle consumer. This property applies only to sequence proxy suppliers, and defaults to zero, which
means that no timeout will be enforced. If the number of events received within a given PacingInterval equals or exceeds MaximumBatchSize, the consumer will receive a sequence of events whose length equals MaximumBatchSize. The data type of this property is TimeBase::TimeT, which is a typedef to an unsigned long long with units of 100s of nanoseconds (10E-7 seconds).

**ConnectionReliability**
This property can be set to either Persistent or BestEffort. If it is set to Persistent for an EventChannel the Notification Service saves the admins, proxy suppliers, and proxy consumers created in that channel in persistent storage. When the Notification Service starts up, it reloads this connection information from persistent storage and uses it to reestablish the connections that were active when it last ran.

Setting ConnectionReliability to Persistent is only valid when Topology Persistence is configured for the Notification Service as described in 27.7.2.

**EventReliability**
This property can be set for a channel or for individual events. It can be set to either BestEffort or Persistent. When it is set to Persistent for a channel, events delivered through that channel will be delivered reliably, even if the Notification Service or a consumer fails and must be restarted.

Events delivered through a reliable channel are delivered reliably unless the EventReliability property for the event is explicitly set to BestEffort.

An unreliable channel is one for which the EventReliability property is not specified, or explicitly set to BestEffort. The EventReliability property for an event delivered through an unreliable channel is ignored.

Event reliability is only available when Event Persistence is configured for the Notification Service as described in 27.7.3.

The EventReliability property should be set to Persistent only when the ConnectionReliability property is also set to Persistent. This is not checked by the current implementation of the TAO Notification Service, but a check may be added in the future.
27.3.5.3 Notification Administration Properties

The following specification-defined QoS properties apply only to an EventChannel. They are established using the set_admin() operation and accessed using the get_admin() operation.

Note

The set_qos() operation is used to set most properties, but these additional QoS properties are supported only by the EventChannel interface and use the set_admin() operation, instead. This distinction is important, because setting an admin property using set_qos() will appear to work, but will have no effect.

MaxQueueLength

This property specifies the maximum number of events the notification event channel will queue internally before it starts discarding events. The events will be discarded according to the DiscardPolicy QoS parameter or RejectNewEvents property. The data type for this property is long.

MaxConsumers

This property defines the maximum number of consumers that can be connected to the notification event channel at a time. If this number is exceeded, then an IMP_LIMIT exception is raised. You must be careful to correctly disconnect consumers when using this property to avoid reaching the limit due to inactive consumers remaining attached to the channel. The data type for this property is long.

MaxSuppliers

This property defines the maximum number of suppliers that can be connected to the notification event channel at a time. If this number is exceeded, then an IMP_LIMIT exception is raised. You must be careful to correctly disconnect suppliers when using this property to avoid reaching the limit due to inactive suppliers remaining attached to the channel. The data type for this property is long.

Note

TAO’s Notification Service implementation differs from the specification in that it raises the CORBA::IMP_LIMIT exception instead of the CosNotifyChannelAdmin::AdminLimitExceeded exception in the cases described above.
RejectNewEvents
This property specifies how the event channel should handle events when the number of events exceeds the value associated with the MaxQueueLength property. When RejectNewEvents is set to true, any attempt to push new events to the channel will result in the IMP_LIMIT CORBA system exception being raised. When this property is set to false, any attempt to push new events to the channel will result in a queued event being discarded according to the value of the DiscardPolicy property. The data type for this property is boolean.

27.3.5.4 TAO-Specific RT CORBA Properties
The following TAO-specific RT CORBA QoS properties are established using the set_qos() operation and accessed using the get_qos() operation. Both of the properties defined here define the allocation of threading resources for proxies. Supplier proxies use these threading resources to dispatch events to consumers. Consumer proxies use them to process incoming events. The threading architecture of the notification channel and related configuration options are discussed in more detail in 27.7.1. Please note that the usage and behavior of these options is dependent on the Real-Time Notification features described in 27.3.8 and are affected by whether the associated library is loaded.

ThreadPool
This property defines a thread pool that a proxy allocates and uses for processing events. It can be set on channel or admin objects, but mainly affects the proxies created under those objects. Setting this property for a consumer proxy overrides the default number of threads defined by the -SourceThreads option and setting it for a supplier proxy overrides the default number of threads defined by the -DispatchingThreads option. The data type of this property is a structure named ThreadPoolParams which is defined in $TAO_ROOT/orbsvcs/orbsvcs/NotifyExt.idl. When used without the RT Notification library, all of the fields of this structure are ignored except static_threads which is an integer value used as the number of threads to allocate for that proxy’s thread pool. When used with the RT Notification library, all of the fields are used to initialize an RT CORBA thread pool and other RT CORBA policies for the POA used to activate the proxy object. See 27.3.8 for further details on using this property with the RT CORBA features.
27.3 Notification Service Features

ThreadPoolLanes
This property defines a thread pool with lanes that a proxy allocates and uses for processing events. It can be set on channel or admin objects, but mainly affects the proxies created under those objects. Setting this property for a consumer proxy overrides the default number of threads defined by the -SourceThreads option and setting it for a supplier proxy overrides the default number of threads defined by the -DispatchingThreads option. The data type of this property is a structure named ThreadPoolLanesParam which is defined in $TAO_ROOT/orbsvcs/orbsvcs/NotifyExt.idl. This property requires the loading of the RT Notification library and results in an exception when used without this library. When used with the RT Notification library, the fields are used to initialize an RT CORBA thread pool with lanes and other RT CORBA policies for the POA used to activate the proxy object. See 27.3.8 for further details on using this property with the RT CORBA features.

Note
When the Notification Service creates threads for a thread pool, it specifies a default priority as part of the thread creation parameters. On HP-UX version 10 and later, the process owner must be a member of a group that has the RTSCHED privilege in order to specify a priority for a new thread. Without this privilege, the thread creation operation will return an error (EPERM). Other operating systems do not exhibit this behavior.

On HP-UX, you can add the RTSCHED privilege to all members of a group with the setprivgrp (1M) command. You must be super-user to execute this command.

For example, for a group named "corbausers", the command would be entered as follows:

```
# setprivgrp corbausers RTSCHED
```

The user that starts the Notification Service should be a member of this group.

By default, the effects of the setprivgrp command are lost after a reboot. See <http://www.faqs.org/faqs/hpux-faq/> for information on how to ensure the privilege group changes become permanent.
27.3.5.5 Other TAO-Specific Properties

BlockingPolicy
Use this property to set a blocking timeout for use when the notification channel’s queue is full. The default behavior, when this property is not specified, is to discard an event when the queue is full (based on the values of RejectNewEvents and DiscardPolicy). When this property is used, the channel blocks for the specified timeout while waiting for the queue to have space for the event. If the timeout expires an event is discarded as in the default case. The data type of this property is TimeBase::TimeT, which is a typedef to an unsigned long long with units of 100s of nanoseconds (10E-7 seconds).

You must use multithreaded dispatching when you specify this option (specified via -DispatchingThreads). In addition, to avoid deadlocks when using this property, be sure to configure the ORB so that dispatching threads do not process incoming events. Here is a suitable service configurator file for a notification channel process that uses the BlockingPolicy property:

```
static Client_Strategy_Factory "-ORBWaitStrategy rw -ORBTransportMuxStrategy exclusive -ORBConnectStrategy blocked"
static Resource_Factory "-ORBFlushingStrategy blocking"
static Notify_Default_Event_Manager_Objects_Factory "-DispatchingThreads 1"
```

27.3.5.6 Accessing and Modifying QoS Properties

The QoS properties mentioned above can be set at the following levels: event channel, admin objects, proxy objects and structured event. If a property is set at more than one level, the property value set at the lower level applies. It is your responsibility to apply these policies in a meaningful way considering the asynchronous nature of event communication.

Table 27-1 indicates the levels at which each property can be set.

<table>
<thead>
<tr>
<th>Property</th>
<th>Event</th>
<th>Proxy</th>
<th>Admin</th>
<th>Channel (QoS)</th>
<th>Channel (Admin)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockingPolicy</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>ConnectionReliability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DiscardPolicy</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
The following code fragment shows how to set the `OrderPolicy` and `DiscardPolicy` properties:

```cpp
CosNotification::QoSProperties qos(2);
qos.length(2);
qos[0].name = CORBA::string_dup (CosNotification::OrderPolicy);
qos[0].value <<= CosNotification::FifoOrder;
qos[1].name = CORBA::string_dup (CosNotification::DiscardPolicy);
qos[1].value <<= CosNotification::LifoOrder;

// qos_admin can be any object whose interface derives from
// CosNotification::QoSAdmin, including EventChannel, ConsumerAdmin,
// SupplierAdmin, ProxyConsumer, ProxySupplier, etc.
qos_admin->set_qos(qos);
```

### Setting QoS in a Structured Event Header
As mentioned previously, you can also insert QoS properties into the variable header fields of a `StructuredEvent`, as follows:

```cpp
CosNotification::StructuredEvent event;
```
// Populate the event’s fixed header fields.
CosNotification::FixedHeader& fh = event.header.fixed_header;
fh.event_type.domain_name = CORBA::string_dup("Messenger");
fh.event_type.type_name = CORBA::string_dup("message");
fh.event_name = CORBA::string_dup("a_message");

// Populate the event’s variable header fields with the qos properties.
CosNotification::OptionalHeaderFields& vh = event.header.variable_header;
vh.length(1);
vh[0].name = CORBA::string_dup(CosNotification::Timeout);
// TimeT is in 10ths of a microsecond (100 nanoseconds).
const TimeT one_minute = (TimeBase::TimeT)(60 * 1000 * 1000 * 10);
vh[0].value <<= one_minute;

27.3.5.7 Negotiating QoS and Conflict Resolution

TAO’s implementation of the Notification Service does not support the validate_qos() and validate_event_qos() operations, and it does not always raise UnsupportedQoS exceptions when expected, you are advised to ensure QoS properties are valid before they are set.

get_qos() Operation
This operation returns the current QoS properties that are set on a given object, including properties that were set by default. The returned properties may even include invalid properties that are ignored by the Notification Service. You can invoke the get_qos() operation as follows:

QoSProperties_var props = qos_admin->get_qos();

27.3.5.8 QoS-Related Exceptions

UnsupportedQoS Exception
Some operations described above may raise this exception if parameters passed to it are QoS properties that are not supported. The exception contains a sequence of erroneous QoS properties and their value ranges. Each included property has an associated error. Table 27-2 describes possible values for the
UnsupportedQoS error codes. It is taken from the Notification Service specification.

Table 27-2 UnsupportedQoS Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSUPPORTED_PROPERTY</td>
<td>This property is not supported by this implementation for this type of target object.</td>
</tr>
<tr>
<td>UNAVAILABLE_PROPERTY</td>
<td>This property cannot be set at (to any value) in the current context (i.e., in the context of other QoS properties).</td>
</tr>
<tr>
<td>UNSUPPORTED_VALUE</td>
<td>The value requested for this property is not supported by this implementation for this type of target object. A range of values which would be supported is returned.</td>
</tr>
<tr>
<td>UNAVAILABLE_VALUE</td>
<td>The value requested for this property is not supported in the current context. A range of values which would be supported is returned.</td>
</tr>
<tr>
<td>BAD_PROPERTY</td>
<td>This property name is unrecognized. The implementation knows nothing about it.</td>
</tr>
<tr>
<td>BAD_TYPE</td>
<td>The type supplied for the value of this property is incorrect.</td>
</tr>
<tr>
<td>BAD_VALUE</td>
<td>An illegal value is supplied for this property. A range of values which would be supported is returned.</td>
</tr>
</tbody>
</table>

BAD_QOS System Exception

This exception is raised during transmission of a structured event from a supplier to the notification event channel when the QoS properties indicated in the header of the event cannot be satisfied by the channel.

27.3.6 Connection Reliability

Normally when the Notification Service starts executing, it creates an Event Channel Factory. Based on command line options it may also create an Event Channel. After that it waits for clients (consumers and suppliers) to create and initialize event channels, admins, proxies, filters, etc. required to deliver events. The collection of all of these objects created inside the notification service is called the notification service topology.

If connection reliability is configured for the Notification Service it will save this topology information, including information about the connections to clients, in persistent storage. When it restarts, it can reload this information, reestablish the connections to the clients (if they are still available), and be...
Notification Service

ready to deliver events without the need for the clients to reconfigure the topology.

To enable connection reliability, several things must happen:

• Topology persistence must be configured via the Service Configurator. For more details on configuring topology persistence, see 27.7.2.
• Client reconnection to the Notification Service must be enabled. For details see 27.7.1.2.
• The ConnectionReliability QoS property must be set to Persistent for those portions of the topology that should be saved.
• The clients must be written to take advantage of the ability to reconnect to existing objects in the Notification Service. Among other things this may involve registering with the Notification Service’s reconnection registry. See 27.3.6.1 for details.

Each portion of the topology inherits the ConnectionReliability property from its parent so setting ConnectionReliability to Persistent for an event channel automatically makes all admins and proxies contained in that channel reliable.

ConnectionReliability may be disabled at the Admin or Proxy level by specifying the ConnectionReliability property as BestEffort for the object that should not be saved. The converse, enabling ConnectionReliability for an Admin in a non-reliable EventChannel or a Proxy in a non-reliable Admin, is not possible. Unless the parent is saved, the child object cannot be saved.

TAO’s implementation of the Notification Service is designed to allow the persistent topology to be reloaded on another computer, even if it has a different architecture or operating system.

27.3.6.1 The Reconnection Registry

Specifying Connection Reliability and configuring persistent topology support provides a basic level of connection reliability. Clients that desire a greater ability to recover their connection after a Notification Service restart can register to receive notification of the restart using a new, TAO-specific feature called the Reconnection Registry.

The EventChannelFactory in TAO implements the following IDL interface:
module NotifyExt
{
    /**
     * \rief An interface that handles registration of suppliers and consumers.
     * 
     * This registry should be implemented by an EventChannelFactory and
     * will call the appropriate reconnect methods for all ReconnectionCallback
     * objects registered with it.
     */
    interface ReconnectionRegistry
    {
        typedef long ReconnectionID;
        ReconnectionID register_callback(in ReconnectionCallback reconection);

        void unregister_callback (in ReconnectionID id);

        /// Check to see if the ReconnectionRegistry is alive
        boolean is_alive ();
    }
}

Clients that wish to use the Reconnection Registry must implement the
following interface:

module NotifyExt
{
    /**
     * \rief An interface which gets registered with a ReconnectionRegistry.
     * 
     * A supplier or consumer must implement this interface in order to
     * allow the Notification Service to attempt to reconnect to it after
     * a failure. The supplier or consumer must register its instance of
     * this interface with the ReconnectionRegistry.
     */
    interface ReconnectionCallback
    {
        /// Perform operations to reconnect to the Notification Service
        /// after a failure.
        void reconnect (in Object new_connection);

        /// Check to see if the ReconnectionCallback is alive
        boolean is_alive ();
    }
}

Clients should narrow the reference the EventChannelFactory to a
NotifyExt::ReconnectionRegistry and then call the
register_callback() operation. After doing so, they will receive a call to their reconnect() operation when the Notification Service has restarted and is ready for use.

27.3.7 Event Reliability

The default behavior of the notification service is to provide best-effort delivery of events. If anything goes wrong delivering an event to a particular consumer, events for that consumer are silently discarded to keep the overall system operating. This behavior can be changed to provide reliable event delivery by configuring event persistence in the Service Configurator (see 27.7.3) and specifying the Persistent setting for the EventReliability QoS property as necessary.

For event reliability to work, persistent EventReliability must be set at the EventChannel level. It may be disabled on a per-event basis by specifying BestEffort for the EventReliability property of a given event. This works for Structured or Sequence events, but not for Any events which have no property settings.

Notice that it is not possible to enable EventReliability for an event unless it is being sent through a reliable EventChannel. Any attempt to do so is silently ignored.

Event reliability also depends on connection reliability (see 27.3.6). If event persistence is configured, but topology persistence is not, the Notification Service will not start.

TAO’s implementation of the Notification Service is designed to allow the persistent event information to be reloaded on another computer, even if it has a different architecture or operating system.

27.3.7.1 The Impact of Using Event Reliability

Setting event reliability for an event causes the event and information about its delivery to be written to an event persistence file. When the notification service is restarted, it first reloads topology information from the topology persistence file and uses this to recreate the internal structures that support event delivery (Channels, Admins, Proxies, and other objects). It then reads any undelivered events from the event persistence file and reactivates them for delivery to the appropriate consumers. As soon as the consumers reconnect to
the notification service, they begin to receive the events that were in route at the time the notification service stopped running.

A supplier that pushes a reliable event to the notification service will not receive a response to the `push()` invocation until the event has been safely stored in the event persistence file.

Reliable events that are being delivered to a consumer that becomes unavailable are held until the connection is reestablished to the consumer or the consumer is restarted (on the same, or a different computer).

### 27.3.8 Real-Time CORBA Support

Applications utilizing the Real-Time CORBA features described in Chapter 9 encounter several challenges when they wish to use the Notification Service. In general, the service needs to honor their quality of service requirements as events pass through the notification channel. TAO supports these applications by bundling a set of related features in the `TAO_RT_Notification` library. These features include:

- New QoS properties to support
  - RT CORBA thread pool usage in the notification channel
  - Specifying the RT CORBA priority model of the notification channel
- Optimized event processing
- Optimized collocation between proxies

### 27.3.8.1 Enabling RT CORBA Support

In order to use the RT Notification features you need to load the `TAO_RT_Notification` library and then specify your configuration via the `ThreadPool` and/or `ThreadPoolLanes` properties. Typically, the library is loaded via a dynamic service configurator directive such as this:

```c
dynamic TAO_Notify_Service Service_Object *
TAO_RT_Notification: _make_TAO_RT_Notify_Service () ""
```

This factory is derived from the normal `CosNotify_Service` factory and takes all the same options as described in 27.7.1 (although not all are necessary or applicable when using the `RT_Notify_Service` factory).
### 27.3.8.2 Thread Pool Property

The **ThreadPool** property contains a **ThreadPoolParams** structure as its data element. Here is the relevant IDL:

```idl
module NotifyExt
 {
 typedef short Priority;
 const Priority minPriority = 0;
 const Priority maxPriority = 32767;

 enum PriorityModel
 {  
  CLIENT_PROPAGATED,
  SERVER_DECLARED
  };
 struct ThreadPoolParams
 {  
  PriorityModel priority_model;
  Priority server_priority;

  unsigned long stacksize;
  unsigned long static_threads;
  unsigned long dynamic_threads;
  Priority default_priority;
  boolean allow_request_buffering;
  unsigned long max_buffered_requests;
  unsigned long max_request_buffer_size;
  
  };

};
```

The members of this structure are used to create two POA policies for a new POA that is used to activate enclosed objects. The **PriorityModelPolicy** uses the **priority_model** and **server_priority** members to define the RT CORBA priority model that the POA uses (server-declared or client-propagated). The remaining members are used to construct a thread pool and **ThreadPoolPolicy** for that POA. Additional details on these POA policies and their usage is available in 9.3.7 and 9.3.8.

When this property is applied on an event channel, the event channel uses this POA (and its thread pool) for all enclosed admin and proxy objects. When applied to an admin object, the admin object uses the POA (and its thread pool) for all enclosed proxy objects.
27.3.8.3 Thread Pool Lanes Property

The ThreadPoolLanes property contains a ThreadPoolLanesParams structure as its data element. Here is the relevant IDL:

```IDL
module NotifyExt
{
  struct ThreadPoolLane
  {
    PriorityModel priority_model;
    Priority server_priority;
    Priority lane_priority;
    unsigned long static_threads;
    unsigned long dynamic_threads;
  };

  typedef sequence <ThreadPoolLane> ThreadPoolLanes_List;

  struct ThreadPoolLanesParams
  {
    PriorityModel priority_model;
    Priority server_priority;
    unsigned long stacksize;
    ThreadPoolLanes_List lanes;
    boolean allow_borrowing;
    boolean allow_request_buffering;
    unsigned long max_buffered_requests;
    unsigned long max_request_buffer_size;
  };
};
```

Most of this is identical to the ThreadPool property with the exception that we are now specifying a thread pool with multiple lanes in place of a monolithic thread pool. Each lane runs at a specific priority and processes events of that priority. Refer to 9.3.7 for details about thread pools with lanes.

27.4 Using the Notification Service

We now look at an example that illustrates the use of Notification Service features. A basic example using structured events is introduced in 27.4.2. In 27.4.3, we discuss how consumer and supplier connections can be managed. We extend the basic example in 27.4.4 by adding offer publication to the supplier and event type subscription to the consumer. Filtering is added to the
supplier and consumer in 27.4.5. In 27.4.6, we add QoS properties to the example. In 27.4.7, we show how to transmit batched events. Finally, in 27.4.8, we show how you can collocate a notification event channel in the same address space as a supplier. For the sake of clarity, error checking in these examples has been kept to a minimum. Full source code for these examples is in the TAO 1.4a source code distribution in subdirectories under $TAO_ROOT/DevGuideExamples/NotifyService/.

27.4.1 Building Notification Service Applications
Table 27-3 lists the set of libraries containing TAO’s Notification Service implementation.

Table 27-3 TAO Notification Service Libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAO_CosNotification</td>
<td>Client-side IDL-generated C++ code (stubs)</td>
</tr>
<tr>
<td>TAO_CosNotification_Skel</td>
<td>Server-side IDL-generated C++ code (skeletons)</td>
</tr>
<tr>
<td>TAO_CosNotification_Serv</td>
<td>Notification Service server implementation</td>
</tr>
<tr>
<td></td>
<td>(required for processes that contain notification channels)</td>
</tr>
<tr>
<td>TAO_CosNotification_Persist</td>
<td>Persistence-related features (required for notification channel server processes that set ConnectionReliability and/or EventReliability to Persist)</td>
</tr>
<tr>
<td>TAO_RT_Notification</td>
<td>Features related to Real-Time CORBA (required for notification channel server processes that use RT CORBA with the Notification Service)</td>
</tr>
</tbody>
</table>

Most processes that use TAO’s Notification Service must link with the TAO_CosNotification and TAO_CosNotification_Skel libraries. Processes that do not implement any notification-related servants (such as some push suppliers) may be able to only link the TAO_CosNotification library. Processes that contain event channel factories and event channels must also link TAO_CosNotification_Serv. These processes must also link either TAO_CosNotification_Persist or TAO_RT_Notification if they use the persistence or real-time features, respectively.

MPC projects for processes that use the Notification Service can simply inherit from the notify_skel base project. MPC projects for notification channel servers can simply inherit from the notify_serv base project. For
example, here is the mpc file for the Notification Service example in
$TAO_ROOT/DevGuideExamples/Notification/Messenger (discussed
in the next section):

project(*Server): namingexe, portableserver, notification_skel {
   requires += exceptions
   Source_Files {
      StructuredEventSupplier_i.cpp
      MessengerServer.cpp
      Messenger_i.cpp
   }
}

project(*Client): namingexe, notification {
   requires += exceptions
   Source_Files {
      MessengerC.cpp
      MessengerClient.cpp
   }
}

project(*Consumer): namingexe, portableserver, notification_skel {
   requires += exceptions
   IDL_Files {
   }
   Source_Files {
      MessengerConsumer.cpp
      StructuredEventConsumer_i.cpp
   }
}

For more information on MPC, see Chapter 4.

27.4.2 A Basic Example
Our example extends the Messenger example discussed in Chapter 3 to use the
Notification Service. Suppose, for example, that the MessengerServer is to
publish the messages it receives so that other interested consumers can receive
them. One way to do this is for the MessengerServer to become a client of
another set of CORBA objects that are interested in receiving the messages
sent to it. In this scenario, the MessengerServer obtains an object reference
for each interested object, then invokes an operation on each object to forward
each message it receives. However, this approach is inflexible and inefficient
because:
The \textit{MessengerServer} must know about all objects that are interested in receiving the messages \textit{a priori}, thereby making it difficult to dynamically add or remove them at run time.

The \textit{MessengerServer} must spend processing time forwarding the messages to these interested objects.

The asynchronous style of event communication used by the Notification Service addresses these issues. It allows suppliers to send messages (events) to consumers that are interested in receiving them, yet neither the consumers nor the suppliers need to know about one another. Moreover, it allows applications to dynamically add or remove suppliers and consumers without impacting other objects in the system. Finally, it transfers responsibility for dispatching events to consumers from suppliers to the notification channel rather than imposing this processing overhead on the suppliers themselves.

We now modify our \textit{MessengerServer} example to send an event to consumers via a notification channel each time it receives a message from a client so that other objects interested in receiving this event can subscribe to the notification channel and receive it. Full source code for this example is in the TAO 1.4a source code distribution in the directory 
$\texttt{TAO_ROOT/DevGuideExamples/NotifyService/Messenger/}$.

27.4.2.1 Starting the Notify\_Service Server
This example uses the \textit{Notify\_Service} and \textit{Naming\_Service} servers in TAO. By default, the \textit{Notify\_Service} server creates a single notification channel factory object and binds it in the root naming context of the Naming Service to the name \texttt{NotifyEventChannelFactory}. The notification channel factory is used to create notification channels. The Naming Service and Notification Service servers must be started in the following order before running this example:

\begin{verbatim}
$\texttt{TAO_ROOT/orbsvcs/Naming\_Service/Naming\_Service}
$\texttt{TAO_ROOT/orbsvcs/Notify\_Service/Notify\_Service}
\end{verbatim}

Command line options for the \textit{Notify\_Service} server are covered in 27.6.
27.4.2.2 Implementing the Structured Push Supplier Interface

The CosNotifyComm::StructuredPushSupplier IDL interface is implemented by our supplier class StructuredEventSupplier_i. The StructuredPushSupplier interface contains the following two operations:

- `disconnect_structured_push_supplier()` allows the notification channel to inform the supplier that it has been disconnected from the channel. Typically, this is called when the channel is destroyed.
- `subscription_change()` allows the notification channel to inform the supplier of changes to the event types its connected consumers are interested in receiving.

```cpp
#include <orbsvcs/CosNotifyCommS.h>

class StructuredEventSupplier_i :
   public virtual POA_CosNotifyComm::StructuredPushSupplier
{
   public:
      // Constructor
      StructuredEventSupplier_i(CORBA::ORB_ptr orb);

      // Override operations from StructuredPushSupplier interface.
      virtual void disconnect_structured_push_supplier ()
         throw (CORBA::SystemException);
      virtual void subscription_change (const CosNotification::EventTypeSeq& events_added,
                                           const CosNotification::EventTypeSeq& events_removed)
         throw (CORBA::SystemException,
                CosNotifyComm::InvalidEventType);

   private:
      CORBA::ORB_var orb_;  
};
```

The `StructuredEventSupplier_i` constructor simply duplicates the ORB reference passed to it and stores it in its data member.

```cpp
StructuredEventSupplier_i::StructuredEventSupplier_i(CORBA::ORB_ptr orb)
   : orb_(CORBA::ORB::_duplicate(orb)) { }
```

The `disconnect_structured_push_supplier()` operation is called when the supplier is being disconnected from the notification channel. Our implementation deactivates the supplier object.

```cpp
void StructuredEventSupplier_i::disconnect_structured_push_supplier ()
```
throw (CORBA::SystemException)
{
    CORBA::Object_var obj = orb_->resolve_initial_references("POACurrent");
    PortableServer::Current_var current =
        PortableServer::Current::_narrow(obj.in());
    PortableServer::POA_var poa = current->get_POA();
    PortableServer::ObjectId_var objectId = current->get_object_id();
    poa->deactivate_object(objectId.in());
}

The subscription_change() operation is called by the supplier’s consumer proxy object to inform the supplier of changes to the subscription information by the notification channel’s consumers. Its implementation does nothing for now, but will be expanded later when we address offers and subscriptions.

void StructuredEventSupplier_i::subscription_change(
    const CosNotification::EventTypeSeq&,
    const CosNotification::EventTypeSeq&
) throw (CORBA::SystemException,
               CosNotifyComm::InvalidEventType)
{
    // More to come...
}

27.4.2.3 Developing the Structured Event Supplier

Next, we modify the implementation of our MessengerServer so that it can behave as a supplier of structured events. Each time it receives a message, it will create a new structured event and populate it with information from the message, then push this event to the notification channel via its consumer proxy.

First, we initialize the ORB and obtain and activate the RootPOA as usual:

#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[]) {
    try {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
        PortableServer::POAManager_var mgr = poa->the_POAManager();
    }
Next, we create a `Messenger_i` servant and activate it in the RootPOA. We then export its object reference as a string and wait for client requests:

```c++
Messenger_i messenger_servant (orb.in());
PortableServer::ObjectId_var oid = poa->activate_object(&messenger_servant);
CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
CORBA::String_var str = orb->object_to_string(messenger_obj.in());
std::ofstream iorFile("Messenger.ior");
iorFile << str.in() << std::endl;
iorFile.close();
std::cout << "IOR written to file Messenger.ior" << std::endl;
orb->run();
orb->destroy();
}
catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
}
return 0;
```

We now turn our attention to the `Messenger_i` class that implements the Messenger interface. Our class now acts as an event supplier. We have added an additional `#include` directive to access the Notification Service channel administration definitions. We have also added new fields, one to hold the object reference of our structured proxy push consumer, and another to hold an ORB reference.

```c++
#include "MessengerS.h"
#include <orbsvcs/CosNotifyChannelAdminC.h>

class Messenger_i : public POA_Messenger {
    public:
        // Constructor and destructor.
        Messenger_i (CORBA::ORB_ptr orb);
        virtual ~Messenger_i (){

        // Override operations from the Messenger interface.
        CORBA::Boolean send_message (const char* user_name, const char* subject, char*& message)
            throw (CORBA::SystemException);
```
private:
  CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
  structured_proxy_consumer_;  
  CORBA::ORB_var orb_; 
};

Our modified implementation of the Messenger_i class follows:

#include "StructuredEventSupplier_i.h"
#include "Messenger_i.h"
#include <orbsvcs/CosNotifyChannelAdminC.h>
#include <orbsvcs/CosNotifyCommC.h>
#include <orbsvcs/CosNamingC.h>

In the constructor, we now initialize the ORB. We then use the Naming Service to access the Notification Service’s notification channel factory:

Messenger_i::Messenger_i (CORBA::ORB_ptr orb)
  : orb_ (CORBA::ORB::_duplicate(orb))
{
  try { 
    CORBA::Object_var naming_obj =
      orb_->resolve_initial_references ("NameService");
    CosNaming::NamingContext_var naming_context =
      CosNaming::NamingContext::_narrow(naming_obj.in());

    When we started the Notify_Service server, it bound the notify channel factory in the root naming context of the Naming Service to the name "NotifyEventChannelFactory." We resolve the factory using this name:

    CosNaming::Name name;
    name.length(1);
    name[0].id = CORBA::string_dup("NotifyEventChannelFactory");
    CORBA::Object_var obj = naming_context->resolve(name);

    CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
      CosNotifyChannelAdmin::EventChannelFactory::_narrow(obj.in());
    if (CORBA::is_nil(notify_factory.in())) {
      std::cerr << "Unable to find notify factory" << std::endl;
    }
  }

  We use the notification channel factory to create a new notification channel. The create_channel() operation takes the following parameters as input:
  • The initial QoS values for the channel.
27.4 Using the Notification Service

- The initial administrative property values. This is a sequence of name/value pairs, where each name is a string and each value is a CORBA::Any.

- The ChannelID as an out parameter. The ChannelID identifies the notification channel within the context of the notification event channel factory.

```c++
CosNotifyChannelAdmin::ChannelID id;
CosNotification::QoSProperties initial_qos;
CosNotification::AdminProperties initial_admin;

CosNotifyChannelAdmin::EventChannel_var notify_channel =
    notify_factory->create_channel (initial_qos,
                                   initial_admin,
                                   id);
if (CORBA::is_nil(notify_channel.in())) {
    std::cerr << "Unable to create notification channel" << std::endl;
}
```

There are several ways to provide access to this notification channel. For this example, we bind it in the root naming context of the Naming Service to the name "MessengerChannel":

```c++
name[0].id = CORBA::string_dup("MessengerChannel");
naming_context->rebind(name, notify_channel.in());
```

Alternatively, the supplier can export ChannelID, which could then be used by consumers to get a reference to the notification channel using the get_event_channel() operation of the notification channel factory.

The Notification Service allows any number of SupplierAdmin objects to be associated with a notification channel. Normally each SupplierAdmin object is responsible for creating and managing proxy consumers with a common set of QoS property settings and filter objects. These proxy consumer objects are used by their suppliers to push events onto the notification channel. The new_for_suppliers() operation is used to obtain a SupplierAdmin object. This operation takes the following parameters:

- InterFilterGroupOperator, which can be either AND_OP or OR_OP. For more information on this parameter, see 27.3.3.
- AdminID, an out parameter that identifies the SupplierAdmin object.
CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
    CosNotifyChannelAdmin::OR_OP;
CosNotifyChannelAdmin::AdminID adminid;
CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
    notify_channel->new_for_suppliers (ifgop, adminid);
if (CORBA::is_nil (supplier_admin.in())) {
    std::cerr << "Unable to access supplier admin" << std::endl;
}

For the push supplier to push events onto the notification channel, we must obtain a proxy push consumer object reference. We get a reference to a proxy push consumer object from the SupplierAdmin object by using the obtain_notification_push_consumer() operation. This operation takes two parameters:

- **ClientType** identifies the type of events our supplier will produce, which can be ANY_EVENT, STRUCTURED_EVENT, or SEQUENCE_EVENT.
- **ProxyID** is an out parameter that identifies the proxy object.

In our example, the supplier is producing structured events, so we specify ClientType to be STRUCTURED_EVENT. Note that the return type of the obtain_notification_push_consumer() operation is CosNotifyChannelAdmin::ProxyConsumer. We must narrow it to CosNotifyChannelAdmin::StructuredProxyPushConsumer since our supplier will be producing structured events. The supplier will use this proxy consumer to push structured events to the notification channel:

```c++
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxyConsumer_var proxy_consumer =
    supplier_admin->obtain_notification_push_consumer(
        CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id);

structured_proxy_consumer_ =
    CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
        proxy_consumer.in());
if (CORBA::is_nil(structured_proxy_consumer_.in())) {
    std::cerr << "Unable to obtain structured proxy push consumer" << std::endl;
}
```

We now create an instance of our StructuredEventSupplier_i push supplier servant class. We activate the supplier servant in the RootPOA of our new ORB:
Using the `connect_structured_push_supplier()` operation, we now connect our supplier to the consumer proxy object. This operation takes an object reference to the supplier as a parameter:

```cpp
CORBA::Object_var supplier_obj = poa->id_to_reference(objectId.in());
CosNotifyComm::StructuredPushSupplier_var supplier =
    CosNotifyComm::StructuredPushSupplier::_narrow(supplier_obj.in());
structured_proxy_consumer_->
    connect_structured_push_supplier(supplier.in());
} catch (CORBA::Exception& ex) { 
    std::cerr << ex << std::endl;
}
}
```

Next, we extend the implementation of our Messenger’s `send_message()` operation. In addition to printing information about the message, we create a new structured event and populate it with the contents of the message. The contents of the structured event are shown in 27.3.1. For this example, the event type domain is `OCI_TAO` and the event type name is `examples`. We invoke the `push_structured_event()` operation of the structured push consumer proxy object to push the event to the notification channel:

```cpp
CORBA::Boolean Messenger_i::send_message ( 
    const char* user_name, 
    const char* subject, 
    char*& message) 
    throw (CORBA::SystemException)
{
    std::cerr << "Message from: " << user_name << std::endl;
    std::cerr << "Subject: " << subject << std::endl;
    std::cerr << "Message: " << message << std::endl;

    // Create a structured event.
    CosNotification::StructuredEvent event;

    // Populate the event’s fixed header fields.
    ```
event.header.fixed_header.event_type.domain_name =
  CORBA::string_dup("OCI_TAO");
event.header.fixed_header.event_type.type_name =
  CORBA::string_dup("examples");
event.header.fixed_header.event_name =
  CORBA::string_dup("myevent");

// Populate the event’s filterable body fields.
 event.filterable_data.length (3);
 event.filterable_data[0].name = CORBA::string_dup("Message from:");
 event.filterable_data[0].value <<= (const char *)user_name;
 event.filterable_data[1].name = CORBA::string_dup("Subject:");
 event.filterable_data[1].value <<= (const char *)subject;
 event.filterable_data[2].name = CORBA::string_dup("Message:=");
 event.filterable_data[2].value <<= (const char *)message;

// Push the event to the notification channel.
 structured_proxy_consumer_->push_structured_event(event);
 return 1;

} // close anonymous namespace

This simple example omits two elements that may be necessary in production
applications. Both are related to process shutdown and any cleanup which
would occur if shutdown() was called on the ORB.

• We have left no way to disconnect from the proxy push consumer. This
means our proxy consumer will likely live in the event channel forever
after our process exits. In a production application, you should be sure to
call disconnect_structured_push_consumer() on the
StructuredProxyPushConsumer to allow the event channel to destroy
this unused object.

• We “leak” the servant object. The solution here is that we should use
reference-counted servants, so the POA automatically deletes any servants
when the POA is deleted.

27.4.2.4 Implementing the Structured Push Consumer Interface

The CosNotifyComm::StructuredPushConsumer IDL interface is
implemented by the class StructuredEventConsumer_i. The three
operations in the StructuredPushConsumer interface are as follows:

• push_structured_event() delivers structured events from the
notification channel to the consumer.
27.4 Using the Notification Service

- disconnect_structured_push_consumer() called when the notification channel disconnects the structured push consumer from its associated proxy supplier.

- offer_change() allows the notification channel to inform the consumer of changes to the event types its connected suppliers intend to produce.

```c++
#include <orbsvcs/CosNotifyCommS.h>

class StructuredEventConsumer_i :
public virtual POA_CosNotifyComm::StructuredPushConsumer
{
public:
  // Constructor.
  StructuredEventConsumer_i(CORBA::ORB_ptr orb);

  // Override operations from StructuredPushConsumer interface.
  virtual void push_structured_event(
    const CosNotification::StructuredEvent& event)
    throw (CORBA::SystemException,
        CosEventComm::Disconnected);
  virtual void disconnect_structured_push_consumer()
    throw (CORBA::SystemException);
  virtual void offer_change (  
    const CosNotification::EventTypeSeq& events_added,
    const CosNotification::EventTypeSeq& events_removed)
    throw (CORBA::SystemException,
        CosNotifyComm::InvalidEventType);

private:
  CORBA::ORB_var orb_;}
}

The StructuredEventConsumer_i constructor simply duplicates the ORB reference passed to it and stores it in its data member:

```c++
StructuredEventConsumer_i::StructuredEventConsumer_i(CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb)) { }
```

The push_structured_event() operation is called for each event that matches the consumer's subscription information. Our implementation simply extracts and prints each element from the filterable body fields of the structured event:

```c++
void StructuredEventConsumer_i::push_structured_event(  
  const CosNotification::StructuredEvent& event)
  throw (CORBA::SystemException,
      ...
```
The `disconnect_structured_push_consumer()` operation is called when the consumer is being disconnected from the notification channel. Our implementation deactivates the consumer object from its POA:

```c++
void StructuredEventConsumer_i::disconnect_structured_push_consumer()
throw (CORBA::SystemException)
{
    CORBA::Object_var obj = orb_->resolve_initial_references("POACurrent");
    PortableServer::Current_var current =
        PortableServer::Current::_narrow(obj.in());
    PortableServer::POA_var poa = current->get_POA();
    PortableServer::ObjectId_var objectId = current->get_object_id();
    poa->deactivate_object(objectId.in());
}
```

The `offer_change()` operation is called by the consumer’s supplier proxy object to inform the consumer of changes in the event types offered by the suppliers of the notification channel. Its implementation does nothing for now, but it will be expanded later when we address offers and subscriptions:

```c++
void StructuredEventConsumer_i::offer_change(
    const CosNotification::EventTypeSeq&,
    const CosNotification::EventTypeSeq&
) throw (CORBA::SystemException,
        CosNotifyComm::InvalidEventType)
{
    // More to come...
}
```

### 27.4.2.5 Developing the Structured Event Consumer

Next, we create a `MessengerConsumer` application to find the notification channel created by the supplier, create an instance of our structured push consumer implementation class, connect the consumer to the channel, and process events:
#include "StructuredEventConsumer_i.h"
#include <orbsvcs/CosNotifyChannelAdminC.h>
#include <orbsvcs/CosNotifyCommC.h>
#include <orbsvcs/CosNamingC.h>

int main(int argc, char* argv[]) {
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var naming_obj = 
      orb->resolve_initial_references ("NameService");
    if (CORBA::is_nil(naming_obj.in())) {
      std::cerr << "Unable to find Naming Service" << std::endl;
      return 1;
    }
    CosNaming::NamingContext_var naming_context = 
      CosNaming::NamingContext::_narrow(naming_obj.in());

    When the supplier created the notification channel, it bound it in the Naming Service’s root naming context to the name MessengerChannel. We now resolve the notification channel from the Naming Service:

    CosNaming::Name name;
    name.length (1);
    name[0].id = CORBA::string_dup("MessengerChannel");
    CORBA::Object_var notify_channel_obj = naming_context->resolve(name);
    CosNotifyChannelAdmin::EventChannel_var notify_channel = 
      CosNotifyChannelAdmin::EventChannel::_narrow(notify_channel_obj.in());
    if (CORBA::is_nil(notify_channel.in())) {
      std::cerr << "Unable to find the notification channel" << std::endl;
      return 1;
    }

    Alternatively, we could use the get_event_channel() operation on the notification channel factory to look up the notification channel using the CosNotifyChannelAdmin::ChannelID returned when the supplier created the channel.

    Any number of ConsumerAdmin administration objects are allowed to be associated with a notification channel. Normally each ConsumerAdmin object is responsible for creating and managing proxy suppliers with a common set of QoS property settings and filter objects. These proxy supplier objects push events to their consumers. A ConsumerAdmin object is obtained using the
new_for_consumers() operation. This operation takes the following parameters:

- InterFilterGroupOperator, which can be either AND_OP, or OR_OP. For more information on this parameter, see 27.3.3.
- AdminID, an out parameter that identifies the ConsumerAdmin object.

```cpp
CosNotifyChannelAdmin::AdminID adminid;
CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
    CosNotifyChannelAdmin::OR_OP;
CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin =
    notify_channel->new_for_consumers (ifgop, adminid);
if (CORBA::is_nil (consumer_admin.in())) {
    std::cerr << "Unable to access consumer admin" << std::endl;
}
```

For the push consumer to connect to the notification channel and begin receiving events, we must obtain a proxy push supplier object reference. We get a reference to a proxy push supplier from the ConsumerAdmin object using the obtain_notification_push_supplier() operation. This operation takes two parameters:

- ClientType identifies the type of events our consumer wishes to receive, which can be ANY_EVENT, STRUCTURED_EVENT, or SEQUENCE_EVENT.
- ProxyID is an out parameter that identifies the proxy object.

In our example, the consumer is receiving structured events, so we specify ClientType to be STRUCTURED_EVENT. Note that the return type of the obtain_notification_push_supplier() operation is defined as CosNotifyChannelAdmin::ProxySupplier. We must narrow it to CosNotifyChannelAdmin::StructuredProxyPushSupplier since our consumer will be receiving structured events. The consumer will connect to this proxy supplier to begin receiving structured events from the notification channel:

```cpp
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxySupplier_var proxy_supplier =
    consumer_admin->obtain_notification_push_supplier(
        CosNotifyChannelAdmin::STRUCTURED_EVENT,
        proxy_id);
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var
    structured_proxy_supplier =
    CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
        proxy_supplier.in());
```
if (CORBA::is_nil (structured_proxy_supplier.in())) {
    std::cerr << "Unable to obtain structured proxy push supplier" << std::endl;
    return 1;
}

We now create an instance of our StructuredEventConsumer_i push consumer servant class. We then activate the servant in the RootPOA and obtain its object reference. We then connect our consumer to the proxy push supplier using the connect_structured_push_consumer() operation. This operation takes an object reference to the consumer as a parameter:

StructuredEventConsumer_i consumer_servant(orb.in());
CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_obj.in());
PortableServer::ObjectId_var oid = poa->activate_object(&consumer_servant);
CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
CosNotifyComm::StructuredPushConsumer_var consumer =
    CosNotifyComm::StructuredPushConsumer::narrow(consumer_obj.in());
structured_proxy_supplier->connect_structured_push_consumer(consumer.in());

We also add information about the event types the consumer is interested in receiving to the proxy push supplier using the subscription_change() operation. We add the structured events identified by the domain name “OCI_TAO” and type name “examples”, and remove all other event types from the subscription:

CosNotification::EventTypeSeq added;
CosNotification::EventTypeSeq removed;
added.length(1);
removed.length(1);

added[0].domain_name = CORBA::string_dup("OCI_TAO");
added[0].type_name = CORBA::string_dup("examples");

removed[0].domain_name = CORBA::string_dup("*");
removed[0].type_name = CORBA::string_dup("*");

structured_proxy_supplier->subscription_change(added, removed);

Note the use of the wildcard character “*” (an asterisk) in the domain and type name fields in the sequence of event types to be removed. This indicates a match with all event types in all domains. Therefore, all event types are removed except for the event type that was added. As before, we could also use “%ALL” with the same meaning.
Next, we activate the POA and enter the ORB event loop so we can receive events:

```cpp
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();
orb->run();
} catch (CORBA::Exception& ex) {
    std::cerr << ex << std::endl;
    return 1;
}
return 0;
```

When an event is pushed to the notification channel, the operation `push_structured_event()` will be invoked on our consumer. This consumer process is assumed to run indefinitely and does not bother to call `disconnect_structured_push_supplier()` on its proxy. Real applications should ensure that the disconnect operation is called before the orb is shut down and the process exits. Failure to do so results in “leaking” the proxy object in the notification channel server.

### 27.4.3 Managing Connections

This section briefly describes how consumers and suppliers can manage their connections to the Notification Service.

#### 27.4.3.1 Connecting and Disconnecting Consumers

Our example used the `connect_structured_push_consumer()` operation to connect to its structured proxy push supplier:

```cpp
structured_proxy_supplier->connect_structured_push_consumer (consumer.in());
```

The consumer remains connected to the notification channel until the corresponding `disconnect_structured_push_supplier()` operation is invoked:

```cpp
structured_proxy_supplier->disconnect_structured_push_supplier ();
```

This disconnects the consumer from the notification channel and causes the structured proxy push supplier to be destroyed.
27.4.3.2 Connecting and Disconnecting Suppliers

Suppliers are connected and disconnected using analogous operations on the structured proxy push consumer object:

```c
structured_proxy_consumer->connect_structured_push_supplier (supplier.in ());

// Do all the publication required...

structured_proxy_consumer->disconnect_structured_push_consumer ();
```

TAO’s Notification Service cleans up resources associated with the supplier when disconnect is called. Failure to call disconnect causes these resource to be leaked.

27.4.3.3 Suspending and Resuming Consumer Connections

Consumers that want to stop receiving events for a period of time can use the suspend_connection() operation. Events published by a supplier while the consumer has suspended the connection will be held in a queue until the consumer invokes the resume_connection() operation. The resume_connection() operation will cause any queued events to be delivered to the consumer as delivery returns to normal.

Using these operations is more efficient than alternative approaches. One alternative is to just stay connected to the notification channel and ignore the events as they are pushed to the consumer. Suspending is much more efficient because the supplier stops pushing events to that consumer and events that are sent to a suspended connection are not lost.

Another approach is to disconnect from the event channel, then later reconnect to the same channel. Events that are waiting to be delivered to the consumer, when the consumer disconnects, and events that are sent to the channel after the disconnection, will never be delivered to this consumer. When the consumer reconnects to the channel it will begin receiving new events sent to that channel.

A framework for the use of these operations is as follows:

```c
structured_proxy_supplier->suspend_connection ();

// Do something else for a while...

structured_proxy_supplier->resume_connection ();
```
27.4.3.4 Destroying the Notification Channel

Notification channels consume resources within the process in which they are created. You can invoke the `destroy()` operation to destroy a notification channel and release its resources when it is no longer needed.

```cpp
notify_channel->destroy();
```

When a notification channel is destroyed, all currently connected consumers and suppliers are notified via their respective disconnect operations. This allows consumers and suppliers to clean up any resource they may be holding.

27.4.4 Using Offers and Subscriptions

In this section, we show how to add offers and subscriptions to our simple example. Full source code for this example is in the TAO 1.4a source code distribution in the directory `$TAO_ROOT/DevGuideExamples/NotifyService/OfferSubscriptions/`.

27.4.4.1 Adding Publication of Offers to the Supplier

The collection of event types a supplier produces is called an offer. A supplier can inform consumers of the notification channel about the types of events it will be producing by invoking the `offer_change()` operation. Later, it can invoke the `offer_change()` operation again to add or remove event types from its offer. The notification channel aggregates the offers of all its suppliers so consumers can be informed of what types of events are currently published via the channel. The `offer_change()` operation can be invoked on the supplier’s consumer proxy object or on its `SupplierAdmin` object.

Here, we show how to add publication of offers to the consumer proxy object. The `offer_change()` operation takes two parameters:

- A sequence of event types the supplier will add to its offer (i.e., the event types it will start supplying).
- A sequence of event types the supplier will remove from its offer (i.e., the event types it will no longer supply).
The type of these parameters is `CosNotification::EventTypeSeq`. Each event type in the sequence is a structure containing two strings:

- The `domain_name` identifies the vertical industry domain in which the event is defined (e.g., telecommunications, finance, health-care).
- The `type_name` categorizes the event within the industry domain.

In the following example, we use the `offer_change()` operation to add the structured event type identified by domain name `OCI_TAO` and type name `examples` to our supplier’s offer, and to remove all other event types from the offer:

```c++
// Add one event type to the offer.
CosNotification::EventTypeSeq events_added;
events_added.length (1);
events_added[0].domain_name = CORBA::string_dup ("OCI_TAO");
events_added[0].type_name = CORBA::string_dup ("examples");

// Remove all other event types from the offer.
CosNotification::EventTypeSeq events_to_be_removed;
events_removed.length (1);
events_removed[0].domain_name = CORBA::string_dup ("*");
events_removed[0].type_name = CORBA::string_dup ("*");

// Change the offer.
structured_proxy_consumer->offer_change (events_added, events_removed);
```

Invoking the `offer_change()` operation on the supplier’s consumer proxy object, as in the above example, affects only that supplier’s offer. Its offer will be aggregated with the offers of all the other suppliers of the notification channel and published as the offer of events for the channel as a whole.

On the other hand, invoking the `offer_change()` operation on the supplier administration object, as in the following example, affects the publication of offers from all the suppliers that share the supplier administration object:

```c++
// Change the offer.
supplier_admin->offer_change (events_added, events_removed);
```

Note that this does not necessarily change the actual types of events the suppliers can produce and push onto the channel. Suppliers can still produce any event type whether or not the event type is published in the notification channel’s offer. The `offer_change()` operation only affects the aggregate
of the event types published for the channel (and therefore visible to consumers via the `obtain_offered_types()` operation).

### 27.4.4.2 Adding Subscriptions to the Consumer

The collection of event types in which a consumer is interested is called a *subscription*. A consumer can inform notification channel suppliers about the types of events it requires by invoking the `subscription_change()` operation. Later, it can invoke the `subscription_change()` operation again to add new event types or remove event types from its subscription. The notification channel aggregates the subscriptions of all its consumers so suppliers can be informed of what types of events consumers of the channel require. The `subscription_change()` operation can be invoked on the consumer’s supplier proxy object or on its `ConsumerAdmin` object.

Here, we show how to add a subscription to the notification event consumer. The `subscription_change()` operation takes two parameters:

- A sequence of event types the consumer will *add* to its subscription (i.e., the event types it wants to *start* receiving).
- A sequence of event types the consumer will *remove* from its subscription (i.e., the event types it *no longer* wishes to receive).

The type of these parameters is `CosNotification::EventTypeSeq`, described in 27.4.4.1.

In the following example, we use the `subscription_change()` operation to add the structured event type identified by domain name `OCI_TAO` and type name `examples` to our consumer’s subscription, and to remove all other event types from the subscription:

```cpp
// Add one event type to the subscription.
CosNotification::EventTypeSeq events_added;
    events_added.length (1);
    events_added[0].domain_name = CORBA::string_dup ("OCI_TAO");
    events_added[0].type_name = CORBA::string_dup ("examples");

// Remove all other event types from the subscription.
CosNotification::EventTypeSeq events_removed;
    events_removed.length (1);
    events_removed[0].domain_name = CORBA::string_dup ("*");
    events_removed[0].type_name = CORBA::string_dup ("*");

// Change the subscription
```
Invoking the `subscription_change()` operation on the consumer’s supplier proxy object, as in the above example, affects only that consumer’s subscription. Its subscription will be aggregated with the subscription of all other consumers of the notification channel and published as the subscription of events for the channel as a whole.

On the other hand, invoking the `subscription_change()` operation on the consumer administration object, as in the following example, affects the publication of subscriptions from all the consumers that share the consumer administration object:

```cpp
// Change the subscription.
consumer_admin->subscription_change(events_added, events_removed);
```

Note that this does not necessarily change the actual types of events the consumers can receive from the channel. Consumers can still receive any event type (that passes all filters) whether or not the event type is in the notification channel’s aggregate subscription. However, you may choose to implement the operation `subscription_change()` in your suppliers such that they stop producing events that are no longer required by any consumers of the channel, as indicated in the aggregate of all the consumers’ subscriptions. In that case, the `subscription_change()` operation may in fact change the actual types of events the consumers will receive.

For example, our structured event supplier class might implement the `subscription_change()` operation as follows to determine if we should continue to produce events for the messages our `MessengerServer` receives.

(Assume that the field `produce_message_events_` is a boolean flag indicating whether or not message events are to be produced. For brevity, we are not checking the "$\%ALL" wildcard.)

```cpp
void StructuredEventSupplier_i::subscription_change (
    const CosNotification::EventTypeSeq& events_added,
    const CosNotification::EventTypeSeq& events_removed)
throw (CORBA::SystemException)
{
    // Check if we are to produce "examples" events.

    // Check the list of removed event types.
    int i;
```
for (i=0; i<events_removed.length(); ++i) {
    if (!strcmp(events_removed[i].domain_name, "OCI_TAO") &&
        !strcmp(events_removed[i].type_name, "examples")) {
        produce_message_events_ = 0;
    } else if (!strcmp(events_removed[i].domain_name, "*") &&
        !strcmp(events_removed[i].type_name, "*")) {
        produce_message_events_ = 0;
    }
}

// Check the list of added event types.
for (i=0; i<events_added.length(); ++i) {
    if (!strcmp(events_added[i].domain_name, "OCI_TAO") &&
        !strcmp(events_added[i].type_name, "examples")) {
        produce_message_events_ = 1;
    } else if (!strcmp(events_added[i].domain_name, "*") &&
        !strcmp(events_added[i].type_name, "*")) {
        produce_message_events_ = 1;
    }
}

We could then modify the Messenger servant’s send_message() operation from 27.4.2.3 to check the produce_message_events_ flag before producing an event.

CORBA::Boolean Messenger_i::send_message (
    const char* user_name,
    const char* subject,
    char*& message
) {
    std::cerr << "Message from: " << user_name << std::endl;
    std::cerr << "Subject: " << subject << std::endl;
    std::cerr << "Message: " << message << std::endl;

    if (produce_message_events_ == 1) {
        // Create and send a structured event as before...
    }
    return 1;
}
27.4.4.3 Obtaining Offer and Subscription Information

Suppliers invoke the `obtain_subscription_types()` operation on the proxy consumer to get the list of event types the consumers connected to the notification channel require. Consumers invoke `obtain_offered_types()` on the proxy supplier to get a list of event types the suppliers connected to the notification channel produce. Each of these operations takes an input parameter of type `CosNotifyChannelAdmin::ObtainInfoMode` that controls what information is returned from the operation as well as whether subsequent automatic updates of subscription information (via the supplier operation `subscription_change()` ) or offer information (via the consumer operation `offer_change()` ) are enabled or disabled.

Here is a portion of the IDL definitions within the `CosNotifyChannelAdmin` module showing the definitions of these operations and the `ObtainInfoMode` type:

```idl
module CosNotifyChannelAdmin
{
  enum ObtainInfoMode {
    ALL_NOW_UPDATES_OFF,
    ALL_NOW_UPDATES_ON,
    NONE_NOW_UPDATES_OFF,
    NONE_NOW_UPDATES_ON
  };

  interface ProxyConsumer:
    CosNotification::QoSAdmin,
    CosNotifyFilter::FilterAdmin
  {
    CosNotification::EventTypeSeq obtain_subscription_types (in ObtainInfoMode mode);
  };

  interface ProxySupplier:
    CosNotification::QoSAdmin,
    CosNotifyFilter::FilterAdmin
  {
    CosNotification::EventTypeSeq obtain_offered_types (in ObtainInfoMode mode);
  };
};
```
Table 27-4 lists the possible values for the ObtainInfoMode parameter.

Table 27-4 ObtainInfoMode Parameter Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_NOW_UPDATES_OFF</td>
<td>Returns the current list of subscription (offer) types known by the target proxy consumer (supplier). Subsequent automatic sending of subscription (offer) update information is disabled.</td>
</tr>
<tr>
<td>ALL_NOW_UPDATES_ON</td>
<td>Returns the current list of subscription (offer) types known by the target proxy consumer (supplier). Subsequent automatic sending of subscription (offer) update information is enabled.</td>
</tr>
<tr>
<td>NONE_NOW_UPDATES_OFF</td>
<td>The invocation does not return any data. Subsequent automatic sending of subscription (offer) update information is disabled.</td>
</tr>
<tr>
<td>NONE_NOW_UPDATES_ON</td>
<td>The invocation does not return any data. Subsequent automatic sending of subscription (offer) update information is enabled.</td>
</tr>
</tbody>
</table>

For example, to get a list of all event types currently required by consumers of the channel, and to enable (via subscription_change()) subscription updates, our supplier would invoke obtain_subscription_types() on its consumer proxy object:

```cpp
CosNotification::EventTypeSeq_var event_type_seq =
    structured_proxy_consumer->obtain_subscription_types (
        CosNotifyChannelAdmin::ALL_NOW_UPDATES_ON);
```

To get a list of all event types currently offered by suppliers of the channel, and to disable future offer updates via offer_change(), our consumer would invoke the obtain_offered_types() operation on its supplier proxy object:

```cpp
CosNotification::EventTypeSeq_var event_type_seq =
    structured_proxy_supplier->obtain_offered_types (
        CosNotifyChannelAdmin::ALL_NOW_UPDATES_OFF);
```

Before obtain_offered_types() or obtain_subscription_types() are called, offer and subscription updates are enabled. If you do not wish to receive these updates, you must explicitly disable them as described above.
27.4 Using the Notification Service

27.4.5 Adding Event Filtering
In this section, we add event filtering to our simple example. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/NotifyService/Filtering/.

27.4.5.1 Adding Event Filtering to the Supplier
Suppose we want to ensure that only events matching certain criteria are supplied to the notification channel. We want events not matching the criteria to be discarded. For example, assume we want to allow only those events for which the Subject field exactly matches the string urgent. Recall that we can filter on any field in the optional header part, filterable body part, or remaining body part of the structured event. To accomplish this, we can create a filter object and attach it to the supplier administration object. This filter would be shared by all the suppliers associated with that SupplierAdmin object. To apply a filter to a single supplier, we would attach it to that supplier’s proxy consumer.

Each notification channel provides a filter factory for creating filter objects. Use the default_filter_factory() operation to obtain an object reference to the channel’s filter factory object:

```cpp
CosNotifyFilter::FilterFactory_var filter_factory =
    notify_channel->default_filter_factory();
```

Next, the factory must create a filter object. The factory’s create_filter() operation takes one parameter representing the name of the constraint language the filter object will use to express filtering constraints. The Notification Service specification defines EXTENDED_TCL as the standard value for specifying ETCL filters. TAO’s Notification Service also understands ETCL, but we’ll use the standard string for portability reasons:

```cpp
CosNotifyFilter::Filter_var filter =
    filter_factory->create_filter("EXTENDED_TCL");
```

Next, create a sequence of constraints. Each constraint is a data structure with two members.
• A sequence of event types indicating the types of events to which the constraint applies. Its type is `CosNotification::EventTypeSeq`, described in 27.4.4.1.

• A string specifying a boolean expression in the chosen constraint grammar that will be applied to the contents of each event if it is of a type contained in the sequence of event types in the first element.

By convention, if the event type sequence has a length of zero, or if it contains a single element in which the domain and event type names are empty strings, the boolean expression applies to all event types. Wildcard characters (“*” and “%ALL”) are also allowed in the event type fields.

Use the `add_constraints()` operation to add the constraints to the filter object. Then add the filter object to the supplier administration object or the proxy consumer object using the `add_filter()` operation.

Here we create a constraint list with a single element. The constraint expression `$.Subject == 'urgent'` is applied to all events regardless of the event’s type.

```cpp
CosNotifyFilter::ConstraintExpSeq constraint_list;
constraint_list.length (1);
constraint_list[0].event_types.length (0);
constraint_list[0].constraint_expr =
    CORBA::string_dup("$.Subject == 'urgent'");
filter->add_constraints (constraint_list);
```

// Apply the filter to all suppliers sharing the same SupplierAdmin object.
supplier_admin->add_filter (filter.in());

or

// Apply the filter to only one supplier’s proxy consumer object.
structured_proxy_consumer->add_filter (filter.in());
```

The result is that only events with a field `Subject` that has the value `urgent` will be forwarded to the notification channel.

The results of multiple constraints within a filter object are ORed together to determine if the event matches any of the filtering constraints. If the result is true, the event has passed the filter. If there are multiple filters added to a proxy or supplier administration object, the event is evaluated against each filter until one of the filters returns true or all the filters return false. If all
filters return false, the event is discarded. If one of the filters returns true, the event passes and is forwarded to the notification channel.

### 27.4.5.2 Adding Event Filtering to the Consumer

Filtering can also be applied to the consumer side of Notification Service event communication. With consumer-side filtering, a structured event consumer can specify the precise set of events it is interested in receiving, based upon filtering criteria. Only events that meet these criteria will be forwarded to the consumer by the consumer’s proxy supplier.

For example, suppose our consumer object wants to receive events only for those messages that originated from "sysadmin@company.com". We can add a filter on the consumer side to filter incoming events accordingly. Similar to what we did in the supplier, we now add a filter object to the consumer administration object.

```cpp
CosNotifyFilter::FilterFactory_var filter_factory =
    notify_channel->default_filter_factory ();

CosNotifyFilter::Filter_var filter =
    filter_factory->create_filter ("EXTENDED_TCL");
if (CORBA::is_nil (filter.in())) {
    std::cerr << "Unable to create filter object" << std::endl;
    return 1;
}

CosNotifyFilter::ConstraintExpSeq constraint_list;
constraint_list.length (1);
constraint_list[0].event_types.length (0);
constraint_list[0].constraint_expr =
    CORBA::string_dup ("$.From == 'sysadmin@company.com'");
filter->add_constraints (constraint_list);

// Apply the filter to all consumers sharing the same ConsumerAdmin object.
consumer_admin->add_filter (filter.in());

or

// Apply the filter to only one consumer’s proxy supplier object.
structured_proxy_supplier->add_filter (filter.in());
```

Our consumer will now receive only those events for messages originating from the **MessengerClient** at **sysadmin@company.com**.
27.4.6 Adding QoS Properties

Next, we extend the MessengerServer code from our basic example to use certain QoS properties. Full source code for this example is in the TAO 1.4a source code distribution in the directory
$TAO_ROOT/DevGuideExamples/NotifyService/QoSProperties/.

We can add notification administrative QoS properties to the event channel created in the Messenger_i class. In this example, we add the MaxQueueLength, MaxSuppliers, and MaxConsumers properties and set their values to 7, 5, and 5 respectively:

```c++
CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
    CosNotifyChannelAdmin::EventChannelFactory::_narrow (obj.in ());

CosNotifyChannelAdmin::ChannelID id;
CosNotification::QoSProperties initial_qos;
CosNotification::AdminProperties initial_admin;
initial_admin.length (3);
initial_admin[0].name =
    CORBA::string_dup (CosNotification::MaxQueueLength);
initial_admin[0].value <<= (CORBA::Long)7;
initial_admin[1].name = CORBA::string_dup (CosNotification::MaxSuppliers);
initial_admin[1].value <<= (CORBA::Long)5;
initial_admin[2].name = CORBA::string_dup (CosNotification::MaxConsumers);
initial_admin[2].value <<= (CORBA::Long)5;

CosNotifyChannelAdmin::EventChannel_var ec =
    notify_factory->create_channel (initial_qos, initial_admin, id);
```

Since we set the MaxQueueLength to be 7, the notification channel will queue the incoming events internally until the number of events exceeds 7. Once the number of events exceeds this limit the channel will start to discard the queued events. We restrict the number of suppliers that can be connected to the notification event channel at a time to be 5. If an attempt is made to connect a supplier when the number of suppliers connected to the channel is 5, IMP_LIMIT exception is raised. We also restrict the number of consumers that can be connected to the notification event channel at a time to be 5. If an attempt is made to connect a consumer when the number of consumers connected to the channel is 5, then IMP_LIMIT exception is raised.
QoS properties can be added at the proxy level too. To illustrate this, we add the `OrderPolicy` property to the proxy supplier object and set its value to `FifoOrder` in our `MessengerConsumer`:

```cpp
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var supplier_proxy;

supplier_proxy =
    CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow (proxy_supplier.in());

CosNotification::QoSProperties properties (1);

properties.length (1);
properties[0].name = CORBA::string_dup (CosNotification::OrderPolicy);
properties[0].value <<= CosNotification::FifoOrder;

supplier_proxy->set_qos (properties);
supplier_proxy->connect_structured_push_consumer(consumer.in());
```

Since we set the `OrderPolicy` property to `FifoOrder`, the proxy supplier will arrange the events in its dispatch queue in the same order as the events were received.

### 27.4.7 Transmitting an EventBatch

The Notification Service also defines interfaces required to transfer more than one structured event using one operation, in the form of a sequence of structured events known as an `EventBatch`. In this section, we extend our example to show how to use batched events. Full source code for this example is in the TAO 1.4a source code distribution in the directory `$TAO_ROOT/DevGuideExamples/NotifyService/EventSequence/`.

We modify the `MessengerServer` to create an `EventBatch` whenever it receives a message from a client and send it through a notification event channel to consumers. To transmit event batches, we need to implement the `SequencePushSupplier` and `SequencePushConsumer` interfaces, which define the behavior of objects that transmits event batches using push-style communication. Our `EventSequenceSupplier_i` class implements the `SequencePushSupplier` interface and our `EventSequenceConsumer_i` class implements the `SequencePushConsumer` interface.
27.4.7.1 Implementing the Sequence Push Supplier Interface

The CosNotifyComm::SequencePushSupplier IDL interface is implemented by our supplier class EventSequenceSupplier_i. The SequencePushSupplier interface has two operations: disconnect_sequence_push_supplier() and subscription_change().

The disconnect_sequence_push_supplier() operation is called when the supplier is being disconnected from the notification channel. Our implementation deactivates the supplier object from its POA:

```c++
void EventSequenceSupplier_i::disconnect_sequence_push_supplier ( )
    throw ((CORBA::SystemException))
{
    CORBA::Object_var obj = orb_->resolve_initial_references ("POACurrent");
    PortableServer::Current_var current =
        PortableServer::Current::_narrow (obj.in());
    PortableServer::POA_var poa = current->get_POA ();
    PortableServer::ObjectId_var objectId = current->get_object_id ();
    poa->deactivate_object (objectId.in());
}
```

The subscription_change() operation is called by the supplier’s consumer proxy object to inform the supplier of changes to the subscription information by the notification channel’s consumers. A specific implementation is not necessary for this example:

```c++
void EventSequenceSupplier_i::subscription_change (const CosNotification::EventTypeSeq& added,
                                                   const CosNotification::EventTypeSeq& removed)
    throw ((CORBA::SystemException, COSNotifyComm::InvalidEventType))
{
}
```

27.4.7.2 Developing the EventBatch Supplier

Next, we implement our MessengerServer as a supplier of an EventBatch. Each time it receives a message, it will create a new EventBatch and populate it with the information from the message, then push this EventBatch to the notification channel via its consumer proxy.

In the MessengerServer class, we initialize the ORB and get a reference to the RootPOA. We create a Messenger_i servant and activate it in the
27.4 Using the Notification Service

RootPOA. Once the servant is activated, we convert the servant’s object reference to a string and write it to the file Messenger.ior as usual:

```cpp
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[]) 
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::narrow(obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    Messenger_i messenger_servant (orb.in());
    PortableServer::ObjectId_var oid =
      poa->activate_object(&messenger_servant);
    CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
    CORBA::String_var str = orb->object_to_string(messenger_obj.in());
    std::ofstream iorfile("Messenger.ior");
    iorfile << str.in() << std::endl;
    std::cout << "IOR written to file Messenger.ior" << std::endl;
    orb->run();
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << ex << std::endl;
    return 1;
  }
  return 0;
}
```

Now we turn our attention to the Messenger_i class that implements the Messenger interface. It now acts as an EventBatch supplier. In the constructor of the Messenger_i class we create a new notification channel and bind it to the root naming context of the Naming Service with the name MessengerChannel:

```cpp
Messenger_i::Messenger_i (CORBA::ORB_ptr orb) :
  orb_(CORBA::ORB::_duplicate(orb))
{
  try {
    CORBA::Object_var naming_obj =
      orb_->resolve_initial_references ("NameService");
    CosNaming::NamingContext_var naming_context =
```
CosNaming::NamingContext::_narrow(naming_obj.in());

CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup("NotifyEventChannelFactory");
CORBA::Object_var obj = naming_context->resolve(name);

CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
    CosNotifyChannelAdmin::EventChannelFactory::_narrow(obj.in());
if (CORBA::is_nil(notify_factory.in())) {
    std::cerr << "Unable to find notify factory" << std::endl;
}

CosNotifyChannelAdmin::ChannelID id;
CosNotification::QoSProperties initial_gos;
CosNotification::AdminProperties initial_admin;

CosNotifyChannelAdmin::EventChannel_var notify_channel =
    notify_factory->create_channel (initial_gos,
        initial_admin,
        id);
if (CORBA::is_nil(notify_channel.in())) {
    std::cerr << "Unable to create notification channel" << std::endl;
}

name[0].id = CORBA::string_dup("MessengerChannel");
naming_context->rebind(name, notify_channel.in());

Once we create the notification channel we obtain a SupplierAdmin object
using new_for_suppliers() operation:

CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
    CosNotifyChannelAdmin::OR_OP;
CosNotifyChannelAdmin::AdminID adminid;
CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
    notify_channel->new_for_suppliers (ifgop, adminid);
if (CORBA::is_nil (supplier_admin.in())) {
    std::cerr << "Unable to access supplier admin" << std::endl;
}

Next, we obtain a proxy push consumer object so that the push supplier can
push events to the notification channel. We obtain a reference to a proxy push
consumer object by invoking obtain_notification_push_consumer() on the SupplierAdmin object. The parameters passed to this operation are
ClientType and ProxyID. ClientType identifies the types of events our
supplier will produce. ProxyID is an out parameter that identifies the proxy
Since our supplier is producing a sequence of structured events, we specify ClientType to be SEQUENCE_EVENT.

We obtain a CosNotifyChannelAdmin::ProxyConsumer object reference from the obtain_notification_push_consumer() operation. We narrow it to CosNotifyChannelAdmin::SequenceProxyPushConsumer since our supplier will be producing sequences of structured events. The supplier uses this proxy consumer to push sequences of structured events to the notification channel:

```
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxyConsumer_var proxy_consumer =
    supplier_admin->obtain_notification_push_consumer(
        CosNotifyChannelAdmin::SEQUENCE_EVENT,
        proxy_id);

sequence_proxy_consumer_ =
    CosNotifyChannelAdmin::SequenceProxyPushConsumer::_narrow(
        proxy_consumer.in());
if (CORBA::is_nil(sequence_proxy_consumer_.in())) {
    std::cerr << "Unable to obtain sequence proxy push consumer" << std::endl;
}
```

Next, we create an instance of our EventSequenceSupplier_i push supplier servant class and activate it in the RootPOA. Finally, we connect our supplier to the consumer proxy object:

```
EventSequenceSupplier_i* supplier_servant =
    new EventSequenceSupplier_i(orb_.in());
CORBA::Object_var poa_obj = orb_->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::narrow(poa_obj.in());
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();
PortableServer::ObjectId_var objectId =
    poa->activate_object(supplier_servant);
CORBA::Object_var supplier_obj = poa->id_to_reference(objectId.in());
CosNotifyComm::SequencePushSupplier_var supplier =
    CosNotifyComm::SequencePushSupplier::_narrow(supplier_obj.in());
sequence_proxy_consumer_->
    connect_sequence_push_supplier(supplier.in());
}
```
In the `send_message()` operation of `Messenger_i` class we print the information about the message received. Then, we create a new structured event and populate it with the contents of the message. For this example, we set the event type domain to be `OCI_TAO` and the event type name to be `examples`. We then create an `EventBatch` from the single structured event we created to illustrate how we can transmit an `EventBatch` without adding much complexity. We invoke the `push_structured_events()` operation of the sequence push consumer proxy object to push the `EventBatch` to the notification channel:

```cpp
CORBA::Boolean Messenger_i::send_message (  
    const char* user_name,  
    const char* subject,  
    char*& message  
)  
{
    std::cerr << "Message from: " << user_name << std::endl;
    std::cerr << "Subject: " << subject << std::endl;
    std::cerr << "Message: " << message << std::endl;

    try
    {
        // Create a structured event.
        CosNotification::StructuredEvent event;

        // Populate the event’s fixed header fields.
        event.header.fixed_header.event_type.domain_name =  
            CORBA::string_dup("OCI_TAO");
        event.header.fixed_header.event_type.type_name =  
            CORBA::string_dup("examples");
        event.header.fixed_header.event_name =  
            CORBA::string_dup("myevent");

        // Populate the event’s filterable body fields.
        event.filterable_data.length (3);
        event.filterable_data[0].name = CORBA::string_dup("Message from:");
        event.filterable_data[0].value <<= (const char *)user_name;
        event.filterable_data[1].name = CORBA::string_dup("Subject:");
        event.filterable_data[1].value <<= (const char *)subject;
        event.filterable_data[2].name = CORBA::string_dup("Message:");
        event.filterable_data[2].value <<= (const char *)message;

        // Create and populate an EventBatch.
        // (We simply put 4 copies of the same event into the sequence.)
        CosNotification::EventBatch events;
        events.length(4);
        events[0] = event;
    }
    catch (const Corba::Exception& e)
    {
        // Exception handling code...
    }
}
```
events[1] = event;
events[2] = event;
events[3] = event;

// Push the event to the notification channel.
sequence_proxy_consumer->push_structured_events(events);
}
catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 0;
}
return 1;
}

27.4.7.3 Implementing the Sequence Push Consumer Interface

The CosNotifyComm::SequencePushConsumer IDL interface is implemented by the class EventSequenceConsumer_i. The SequencePushConsumer interface contains the three operations: push_structured_events(), disconnect_sequence_pushConsumer(), and offer_change().

The push_structured_events() operation is invoked for each event sequence that matches the consumer’s subscription information. Our implementation simply extracts and prints the filterable body fields of the structured events contained in the EventBatch:

```cpp
void EventSequenceConsumer_i::push_structured_events {
    const CosNotification::EventBatch& events
    throw ((CORBA::SystemException, CosEventComm::Disconnected))
    {
        const char* value;
        for (unsigned int n=0; n<events.length(); ++n) {
            for (unsigned int i=0; i<events[n].filterable_data.length(); ++i) {
                events[n].filterable_data[i].value >>= value;
                std::cout << events[n].filterable_data[i].name << "\t" << value << std::endl;
            }
        }
    }
}
```

The disconnect_sequence_pushConsumer() operation is invoked when the consumer is being disconnected from the notification channel. Our implementation deactivates the consumer object from its POA:
void EventSequenceConsumer_i::disconnect_sequence_push_consumer ()
  throw ((CORBA::SystemException))
{
  CORBA::Object_var obj = orb_->resolve_initial_references("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_narrow(obj.in());
  PortableServer::POA_var poa = current->get_POA();
  PortableServer::ObjectId_var objectId = current->get_object_id();
  poa->deactivate_object(objectId.in());
}

The offer_change() operation is invoked by the consumer’s supplier proxy object to inform the consumer of changes in the event types offered by the suppliers of the notification channel. A specific implementation is not necessary for this example:

void StructuredEventConsumer_i::offer_change(
  const CosNotification::EventTypeSeq&,
  const CosNotification::EventTypeSeq&)
  throw (CORBA::SystemException)
{
}

### 27.4.7.4 Developing the Event Consumer

Next, we create a MessengerConsumer application to find the notification channel created by the supplier. In this application we create an instance of the EventSequenceConsumer push consumer class, connect the consumer to the channel, and process events.

In the MessengerConsumer class, we initialize the ORB, locate the Naming Service, then resolve the MessengerChannel that was created by our supplier:

```c++
#include <orbsvcs/CosNamingC.h>
#include <orbsvcs/CosNotifyChannelAdminC.h>
#include <orbsvcs/CosNotificationC.h>
#include <iostream>

int main(int argc, char* argv[])
{
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var naming_obj =
      orb->resolve_initial_references("NameService");
```
if (CORBA::is_nil(naming_obj.in())) {
    std::cerr << "Unable to find Naming Service" << std::endl;
    return 1;
}

CosNaming::NamingContext_var naming_context =
    CosNaming::NamingContext::_narrow(naming_obj.in());

CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup("MessengerChannel");
CORBA::Object_var notify_channel_obj = naming_context->resolve(name);
CosNotifyChannelAdmin::EventChannel_var notify_channel =
    CosNotifyChannelAdmin::EventChannel::_narrow(notify_channel_obj.in());
if (CORBA::is_nil (notify_channel.in())) {
    std::cerr << "Unable to find the notification channel" << std::endl;
    return 1;
}

Next, we obtain a ConsumerAdmin object using the new_for_consumers() operation:

   CosNotifyChannelAdmin::AdminID adminid;
   CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
        CosNotifyChannelAdmin::OR_OP;
   CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin =
        notify_channel->new_for_consumers (ifgop, adminid);
if (CORBA::is_nil (consumer_admin.in())) {
    std::cerr << "Unable to access consumer admin" << std::endl;
}

We then use the obtain_notification_push_supplier() operation to obtain a proxy supplier. The parameters passed to this operation are ClientType and ProxyID. The ClientType parameter identifies the types of events our supplier will produce. The ProxyID parameter is an out parameter that identifies the proxy object. Since our consumer is receiving a sequence of structured events, we specify ClientType to be SEQUENCE_EVENT.

We obtain a CosNotifyChannelAdmin::ProxySupplier object from the obtain_notification_push_supplier() operation. We narrow it to CosNotifyChannelAdmin::SequenceProxyPushSupplier since our supplier will be receiving sequences of structured events. The consumer will connect to this proxy supplier to begin receiving sequences of structured events from the notification channel:
Notification Service

CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxySupplier_var proxy_supplier =
    consumer_admin->obtain_notification_push_supplier(
        CosNotifyChannelAdmin::SEQUENCE_EVENT,
        proxy_id);
CosNotifyChannelAdmin::SequenceProxyPushSupplier_var
    sequence_proxy_supplier =
    CosNotifyChannelAdmin::SequenceProxyPushSupplier::_narrow(
        proxy_supplier.in());
if (CORBA::is_nil (sequence_proxy_supplier.in())) {
    std::cerr << "Unable to obtain sequence proxy push supplier" << std::endl;
    return 1;
}

We create an instance of our EventSequenceConsumer_i push consumer
servant class and activate it in the RootPOA. We then obtain its object
reference and connect it to the proxy push supplier using the
connect_sequence_push_consumer() operation. We pass the
consumer’s object reference as a parameter this operation. Information about
the event types that the consumer is interested in receiving is also passed to the
proxy push supplier using the subscription_change() operation:

EventSequenceConsumer_i consumer_servant(orb.in());
CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa =
    PortableServer::POA::_narrow(poa_obj.in());
PortableServer::ObjectId_var oid =
    poa->activate_object(&consumer_servant);
CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
CosNotifyComm::SequencePushConsumer_var consumer =
    CosNotifyComm::SequencePushConsumer::_narrow(consumer_obj.in());
structured_proxy_supplier->connect_sequence_push_consumer(consumer.in());

CosNotification::EventTypeSeq added;
CosNotification::EventTypeSeq removed;
added.length(1);
removed.length(1);

added[0].domain_name = CORBA::string_dup("OCI_TAO");
added[0].type_name = CORBA::string_dup("examples");

removed[0].domain_name = CORBA::string_dup("*");
removed[0].type_name = CORBA::string_dup("*");

sequence_proxy_supplier->subscription_change(added, removed);
Finally, we activate the POA and enter the ORB event loop so we can receive events. When an event is pushed to the notification channel, the operation `push_structured_events()` will be invoked on our consumer.

```c++
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();
orb->run();
orb->destroy();
}
catch (CORBA::Exception& ex) {
  std::cerr << "Caught a CORBA exception: " << ex << std::endl;
  return 1;
}
return 0;
}

27.4.8 Collocated Notification Channels

The previous examples used the NotifyEventChannelFactory object provided by the Notify_Service server to create notification channels. Thus, the channel factory and the notification channels have all been located in a separate process (the Notify_Service server) from the consumers or suppliers. For performance reasons, there may be situations in which you want to create and manage local, or collocated, notification channels. For example, you may want to collocate a notification channel with a particular supplier to avoid a network hop for each event originating from that supplier. Full source code for this example is in the TAO 1.4a source code distribution in the directory


27.4.8.1 Collocated Notification Channel Example

Here we show how to create and use a local notification channel factory. We then use this factory to create a notification channel collocated with a supplier.

First, include the following header file that defines the notification channel factory implementation (servant) class:

```c++
#include <orbsvcs/Notify/Notify_EventChannelFactory_i.h>
```

Next, in your program’s `main()` function, initialize the ORB and obtain and activate the `RootPOA` as usual:
Notification Service

```c
int main(int argc, char* argv[]) {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var poa_object = orb->resolve_initial_references("RootPOA");

    PortableServer::POA_var poa =
        PortableServer::POA::_narrow(poa_object.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    The TAO_Notify_EventChannelFactory_i::create() operation is now used to create an instance of TAO’s notification event channel factory implementation class. The create() function not only creates a factory servant instance, but also activates it in a POA (provided as a parameter) and returns the factory’s object reference:

    CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
        TAO_Notify_EventChannelFactory_i::create(poa.in());

    Next, we use this local factory to create our notification channel using the create_channel() operation exactly as before (see 27.4.2.3). However, the new notification channel will be a local object:

    CosNotifyChannelAdmin::ChannelID id;
    CosNotification::QoSProperties initial_qos;
    CosNotification::AdminProperties initial_admin;

    CosNotifyChannelAdmin::EventChannel_var notify_channel =
        notify_factory->create_channel(initial_qos,
                                       initial_admin,
                                       id);

    The rest of the program requires no additional changes to use this collocated notification channel.
```

27.4.9 Real-Time Notification Example

In this section, we’ll discuss the modifications to the basic messenger example from 27.4.2 that are necessary to use the RT CORBA features with the Notification Service. We present selected snippets of this code with our additions and changes in a bold font. Full source code for this example is in the TAO 1.4a source code distribution in the directory $TAO_ROOT/DevGuideExamples/NotifyService/RTNotify/.
### 27.4.9.1 Notification Server Configuration

The notification server needs to be started with the RT CORBA and RT Notification libraries loaded. We accomplish this by placing these directives:

```c
#include "TAO_RTORBLoader.h"

int main (int argc, char **argv)
{
  const char *app_name = argv[0];
  CORBA::Environment env;
  TAO::ORBInitGroup::init(argc, argv, env);
  TAO::RTORB::init(env, app_name);
  TAO::Servant::init(env);
  TAO::Ice::IceInit::init(env);
  TAO::Salvo::init(env);

  // Load the RT Notification library
  TAO::.servant::TAO_RT_ORB_Loader_Servant::make_TAO_RT_ORB_Loader();
  TAO::servant::TAO_RT_Notify_Service::make_TAO_RT_Notify_Service();

  // Start the server with the configuration file
  TAO::servant::Notify_Service::Notify_Service_init(argc, argv, null);

  return 0;
}
```

in the `notify.conf` file and launching the server with this configuration file:

```bash
$TAO_ROOT/orbsvcs/Notify_Service/Notify_Service -ORBSvcConf notify.conf
```

This dynamically loads the RT Notification library and enables the RT CORBA-related features in the notification service.

### 27.4.9.2 Messenger Server Changes

The Messenger Server process acts as our event supplier and we modify it here to allocate a thread pool with lanes for our consumer proxy and to set the RT CORBA priority for event publication.

First, we need to set the `ThreadPoolLanes` property on the supplier admin. This is done before creating the consumer proxy:

```c
CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
  ec->new_for_suppliers (ifgop, adminid);

NotifyExt::ThreadPoolLanesParams tpl_params;

tpl_params.priority_model = NotifyExt::CLIENT_PROPAGATED;
tpl_params.server_priority = DEFAULT_PRIORITY;
tpl_params.stacksize = 0;
tpl_params.allow_borrowing = 0;
tpl_params.allow_request_buffering = 0;
tpl_params.max_buffered_requests = 0;
tpl_params.max_request_buffer_size = 0;
.tpl_params.lanes.length (2);
  tpl_params.lanes[0].lane_priority = LOW_PRIORITY;
  tpl_params.lanes[0].static_threads = 2;
  tpl_params.lanes[0].dynamic_threads = 0;
  tpl_params.lanes[1].lane_priority = HIGH_PRIORITY;
  tpl_params.lanes[1].static_threads = 2;
  tpl_params.lanes[1].dynamic_threads = 0;
CosNotification::QoSProperties qos;
```
qos.length(1);
qos[0].name = CORBA::string_dup (NotifyExt::ThreadPoolLanes);
qos[0].value <<= tpl_params;

supplier_admin->set_qos(qos);

CosNotifyChannelAdmin::ProxyID supplieradmin_proxy_id;

CosNotifyChannelAdmin::ProxyConsumer_var proxy_consumer =
supplier_admin->obtain_notification_push_consumer(
  CosNotifyChannelAdmin::STRUCTURED_EVENT,
  supplieradmin_proxy_id);

This causes the supplier admin to create a POA with the Priority Model and
Thread Pool policies that is used to activate any subsequently created
consumer proxies. When we subsequently use our consumer proxy to begin
publishing events they are placed into the correct lanes of the thread pool
based on the priority of the incoming push_structured_event() operation. Here are the modifications related to setting the priority for event
publication:

CORBA::Object_var current_obj =
  this->orb_->resolve_initial_references ("RTCurrent");
RTCORBA::Current_var current = RTCORBA::Current::_narrow (current_obj.in ());

  current->the_priority(HIGH_PRIORITY);

  // Set up the events for publication...
  consumer_proxy_->push_structured_event(event);

Obtaining and narrowing the RTCurren object can be done once for the
process and stored for use in all threads. The stored RTCurren object
reference can then be used to set the priority for each thread where push
operations are called.

27.4.9.3 Messenger Consumer Changes

The consumer side changes involve setting up a thread pool for the supplier
proxy and setting up the proper RT CORBA policies for the consumer object’s
activation.

Setting up the thread pool for the supplier proxy is analogous to what we saw
above for the supplier-side:

CosNotifyComm::StructuredPushConsumer_var consumer =
27.4 Using the Notification Service

CosNotifyComm::StructuredPushConsumer::_narrow (consumer_obj.in ());

NotifyExt::ThreadPoolLanesParams tpl_params;

tpl_params.priority_model = NotifyExt::CLIENT_PROPAGATED;
tpl_params.server_priority = DEFAULT_PRIORITY;
tpl_params.stacksize = 0;
tpl_params.allow_borrowing = 0;
tpl_params.allow_request_buffering = 0;
tpl_params.max_buffered_requests = 0;
tpl_params.max_request_buffer_size = 0;
tpl_params.lanes.length (2);
tpl_params.lanes[0].lane_priority   = LOW_PRIORITY;
tpl_params.lanes[0].static_threads  = 2;
tpl_params.lanes[0].dynamic_threads = 0;
tpl_params.lanes[1].lane_priority   = HIGH_PRIORITY;
tpl_params.lanes[1].static_threads  = 2;
tpl_params.lanes[1].dynamic_threads = 0;
CosNotification::QoSProperties qos;
qos.length(1);
qos[0].name = CORBA::string_dup (NotifyExt::ThreadPoolLanes);
qos[0].value <<= tpl_params;

consumer_admin->set_qos(qos);

CosNotifyChannelAdmin::ProxyID consumeradmin_proxy_id;

CosNotifyChannelAdmin::ProxySupplier_var proxy_supplier =
consumer_admin->obtain_notification_push_supplier(
    CosNotifyChannelAdmin::STRUCTURED_EVENT,
    consumeradmin_proxy_id);

In order for our consumer to be enabled for RT CORBA and honor the
priorities that the events carry through the event channel, we need to create the
necessary RT CORBA policies, construct an RT POA with them, and then use
this POA to activate our consumer. The code below illustrates this for our
eample.

CORBA::Object_var poa_object =
orb->resolve_initial_references("RootPOA");

PortableServer::POA_var poa =
PortableServer::POA::_narrow (poa_object.in());

CORBA::Object_var rtorb_obj = orb->resolve_initial_references("RTORB");
RTCORBA::RTORB_var rt_orb = RTCORBA::RTORB::_narrow (rtorb_obj.in ());

// Create an RT POA with a lane at the given priority.
CORBA::Policy_var priority_model_policy =
    rt_orb->create_priority_model_policy (RTCORBA::CLIENT_PROPAGATED,
    DEFAULT_PRIORITY);

RTCORBA::ThreadPoolLanes lanes (2);
lanes.length (2);

lanes[0].lane_priority = LOW_PRIORITY;
lanes[0].static_threads = 2;
lanes[0].dynamic_threads = 0;
lanes[1].lane_priority = HIGH_PRIORITY;
lanes[1].static_threads = 2;
lanes[1].dynamic_threads = 0;

// Create a thread-pool.
CORBA::ULong stacksize = 0;
CORBA::Boolean allow_request_buffering = 0;
CORBA::ULong max_buffered_requests = 0;
CORBA::ULong max_request_buffer_size = 0;
CORBA::Boolean allow_borrowing = 0;

// Create the thread-pool.
RTCORBA::ThreadPoolId threadpool_id =
    rt_orb->create_threadpool_with_lanes (stacksize,
    lanes,
    allow_borrowing,
    allow_request_buffering,
    max_buffered_requests,
    max_request_buffer_size);

// Create a thread-pool policy.
CORBA::Policy_var lanes_policy =
    rt_orb->create_threadpool_policy (threadpool_id);

CORBA::PolicyList poa_policy_list(2);
poa_policy_list.length (2);
poa_policy_list[0] = priority_model_policy;
poa_policy_list[1] = lanes_policy;

PortableServer::POAManager_var poa_manager = poa->the_POAManager ();

PortableServer::POA_var rt_poa = poa->create_POA ("RT POA",
    poa_manager.in (),
    poa_policy_list);

StructuredEventConsumer_i servant (orb.in());

PortableServer::ObjectId_var objectId =
    rt_poa->activate_object (&servant);
27.5 Compatibility with the Event Service

The Notification Service is backwards compatible with the Event Service. Therefore, existing Event Service applications can interoperate with Notification Service applications. For example, Event Service consumers and suppliers can connect to a Notification Service event channel using the basic IDL interfaces defined by the Event Service.

27.6 Notify_Service Command Line Options

The full path of the Notify_Service server is:

```
$TAO_ROOT/orbsvcs/Notify_Service/Notify_Service
```

The Notify_Service server provides a single notification event channel factory in its own process. By default, it binds the factory to the name NotifyEventChannelFactory in the root naming context of the Naming Service. The Naming Service must already be running to use the Notify_Service server.

The Notify_Service server accepts various command line options to control aspects of its initialization. Table 27-5 describes these command line options.

**Table 27-5 Notify_Service Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-?</td>
<td>Displays the available options.</td>
<td>None</td>
</tr>
</tbody>
</table>
To allow the Notification Service to be used in a wide variety of situations, its behavior can be configured via three separate factories using the service configurator. See 18.3 for a general discussion of service configurator usage. The majority of the notification service configuration options, especially

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Factory</td>
<td>Specifies the name with which to bind the notification channel factory in the root naming context of the Naming Service.</td>
<td>NotifyEventChannelFactory</td>
</tr>
<tr>
<td>factory_name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Boot</td>
<td>Specifies that factory_name should be registered with the IOR table. This allows the use of corbaloc style URLs and also TAO’s Implementation Repository. (See Chapter 30.)</td>
<td>Disabled</td>
</tr>
<tr>
<td>-NameSvc</td>
<td>Binds the notification event channel factory with the Naming Service.</td>
<td>Enabled</td>
</tr>
<tr>
<td>-NoNameSvc</td>
<td>Specifies that the Naming Service should not be used.</td>
<td>Disabled</td>
</tr>
<tr>
<td>-IORoutput</td>
<td>Specifies the name of the file for storing the notification channel factory’s IOR as a string.</td>
<td>stderr</td>
</tr>
<tr>
<td>file_name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Channel</td>
<td>Specifies that a notification channel should be created in addition to the notification channel factory.</td>
<td>Disabled</td>
</tr>
<tr>
<td>-ChannelName</td>
<td>Specifies the name with which to bind the notification channel, if created, in the root naming context of the Naming Service.</td>
<td>NotifyEventChannel</td>
</tr>
<tr>
<td>channel_name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Notify_TPReactor</td>
<td>Specifies the number of worker threads in the thread pool for TP reactor. The ORB should be instructed to use the TP reactor via the service configurator. Note: The Notify_TPReactor option is deprecated and is only included for backwards compatibility</td>
<td>1</td>
</tr>
<tr>
<td>nthreads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ORBRunThreads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nthreads</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 27.7 Notification Service Configuration Options

To allow the Notification Service to be used in a wide variety of situations, its behavior can be configured via three separate factories using the service configurator. See 18.3 for a general discussion of service configurator usage. The majority of the notification service configuration options, especially
threading controls, are available via the Notify Service Factory as discussed in 27.7.1. The topology persistence feature is configured via the Connection Reliability Factory (see 27.7.2) and the event persistence feature is controlled via the Event Reliability Factory (see 27.7.3).

27.7.1 Notify Service Factory Options

The options specified for the notify service factory allow the Notification Service to use multiple threads and other asynchronous techniques to reduce the coupling between the Notification Service and its clients. These options are applied via the service configurator to the Notify Service Factory. This factory was previously called Notify_Default_Event_Manager_Objects_Factory, but has now been mercifully renamed to TAO_CosNotify_Service. You can still use both names in your service configuration files, but we recommend using the latter and use it exclusively in the ensuing examples. The options described in this section can also be passed via the RT Notify Service Factory when it is used to load the RT CORBA Notification features as described in 27.3.8.

By default, the notification channel processes events immediately, or reactively. The thread that processes the incoming publication of a push() operation is also used for all processing of that event, including subsequent push() invocations on the consumers. Immediate processing results in less overhead, but it results in coupling between clients as the behavior of one client can have an adverse effect on other clients. A slow consumer delays delivery of the same event to other consumers and may block publication of other events by the supplier. Various options below add additional thread breaks in the processing pipeline by adding thread pools at different locations.

A notification channel server can operate in one of two basic threading modes. This mode changes the behavior of some of the following options. By default, different proxies (consumer and supplier) share the same server threading resources in the shared resource model. By contrast, when the -AllocateTaskPerProxy option is used, we are using the task per proxy model where each proxy is allocated its own threading resources.

There are two “branch points” during the processing of an event, the Source point and the Dispatch point. At each of these branch points, the event can be processed immediately (on the current thread) or it can be queued for later processing by one thread from a pool of threads. Use the -DispatchingThreads option to specify the number of threads to allocate
at the Dispatch point and the -SourceThreads option to specify the number of threads at the Source point. When separate lookup threads are used, the ORB thread processing a supplier’s push() limits itself to queueing the event for processing on a source thread. When dispatching threads are used, the outgoing push() invocation on the consumer occurs on the dispatching thread. All other processing is done before the dispatching thread break (by either the source or ORB thread). The number of ORB threads allocated can be specified via the -ORBRunThreads option as discussed in 27.6.

When using the task per proxy model, the specified number of source threads are created for each consumer proxy (or supplier) and the specified number of dispatching threads are created for each supplier proxy (or consumer).

The options specified via the service configurator act as the global defaults for all channels and proxies in a process but can be overridden for individual proxies via the ThreadPool and ThreadPoolLanes properties as discussed in 27.3.5.4 and 27.3.8.

Previous versions of the Notification Service allowed for the number of threads to be specified at two additional branch points, Lookup and Listener Evaluation. These extra branch points were previously honored as separate points but are now collapsed into the existing branch points. The Source Evaluation point has been merged into the Lookup point and the Listener Evaluation point has been merged into the Dispatch point. The corresponding -ListenerThreads and -LookupThreads options have been deprecated and should no longer be used. If specified, these options result in a warning message and the threads specified may be added to the remaining branch points. Listener threads are always added to the dispatch thread pool. When the task per proxy model is used, lookup threads are added to the source thread pool. Otherwise, lookup threads are ignored.

Originally, it was necessary to enable multithreading at one of these branch points, then specify the number of threads available. This has been streamlined so that specifying the number of threads automatically enables multithreading. Because of this change the following options, are deprecated, but still recognized and ignored: -MTDispatching, -MTSourceEval, -MTLookup, and -MTListenerEval. You should no longer use these options as they result in warnings in TAO 1.4a and will eventually be removed.

The -AsynchUpdates option, if specified, creates a separate thread responsible for delivering subscription information to clients. Unless you are using subscription information to control what information is generated by
suppliers based on what information the consumer subscriptions, you can ignore this option.

The `- AllowReconnect` option controls whether consumer and supplier proxies allow subsequent connect calls to proxies that are already connected. These options appear in the following line in the service configurator file (e.g., `svc.conf`):

```
static TAO_CosNotify_Service "options"
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-AllocateTaskPerProxy</code></td>
<td>27.7.1.1</td>
<td>Specifies that the Notification Service should allocate thread resources on a per proxy basis.</td>
</tr>
<tr>
<td><code>-AllowReconnect</code></td>
<td>27.7.1.2</td>
<td>Allows consumers and suppliers to reconnect to existing proxies.</td>
</tr>
<tr>
<td><code>-AsynchUpdates</code></td>
<td>27.7.1.3</td>
<td>Causes subscription and publication updates to be sent asynchronously.</td>
</tr>
<tr>
<td><code>-DispatchingThreads nthreads</code></td>
<td>27.7.1.4</td>
<td>Specifies the number of dispatching threads to use.</td>
</tr>
<tr>
<td><code>-ListenerThreads nthreads</code></td>
<td>27.7.1.5</td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>-LookupThreads nthreads</code></td>
<td>27.7.1.6</td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>-MTDispatching</code></td>
<td>27.7.1.7</td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>-MTListenerEval</code></td>
<td>27.7.1.8</td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>-MTLookup</code></td>
<td>27.7.1.9</td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>-MTSourceEval</code></td>
<td>27.7.1.10</td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>-NoUpdates</code></td>
<td>27.7.1.11</td>
<td>Globally disables subscription and publication updates.</td>
</tr>
<tr>
<td><code>-SourceThreads nthreads</code></td>
<td>27.7.1.12</td>
<td>Specify the number of threads to allocate for processing incoming requests.</td>
</tr>
</tbody>
</table>
27.7.1.1 **AllocateTaskPerProxy**

**Description**  This option specifies that the Notification Service should allocate thread resources (i.e., for dispatching and source processing) on a per proxy basis. Using this option affects the behavior of the `-SourceThreads` and `-DispatchingThreads` options as well as some of the QoS Properties.

**Usage**  The notify service factory provides options to enable multithreading and allocate threads for event dispatching and source processing. By default, the requested number of threads is allocated for each consumer and supplier admin object in the notification channel. Different consumer and supplier proxies that are created in that admin object share the same pool of threads.

For example, multithreaded event dispatching is enabled by the `-DispatchingThreads` option (see 27.7.1.4) which also specifies the number of threads allocated for dispatching. By default, the number of threads used for dispatching is whatever is supplied to the `-DispatchingThreads` option for each consumer admin object, regardless of the number of proxies. Use the `-AllocateTaskPerProxy` option to specify that the specified number of threads should be allocated for each supplier proxy. Similarly, this option also alters the source processing threads from being allocated on a per-proxy basis to a per-admin basis.

**Impact**  Allocating threads on a per-proxy basis consumes more resources than allocating them on a per-admin basis. Large numbers of consumers and suppliers can cause creation of excessive numbers of threads.

Events are queued within the notification channel on a per-task basis. This means that when this option is not specified, all consumers share the same event queue and any queue-based QoS properties (DiscardPolicy, OrderPolicy, MaxEventsPerConsumer, BlockingPolicy) function on a per-admin basis. When this option is specified, each proxy applies these QoS properties independently to their own queue.

**See Also**  27.7.1.4, 27.7.1.12

**Example**  static TAO_CosNotify_Service "-AllocateTaskPerProxy -DispatchingThreads 5"
27.7 Notification Service Configuration Options

27.7.1.2 AllowReconnect

**Description**  This option enables the non-standard behavior necessary to allow clients to reconnect to an existing proxy in the notification service. It should be enabled if topology persistence is configured.

**Impact**  According to the OMG specification for the notification service, a client that attempts to connect to a proxy that is already in use should receive a `CosEventChannelAdmin::AlreadyConnected` exception. Unfortunately, this prevents recovery from certain types of failure. When this option is enabled, clients can reconnect to an existing proxy.

Setting this option may allow a rogue client to “steal” a connection that is being used by another client. This should be taken into consideration in designing a reliable system.

**Example**  `static TAO_CosNotify_Service "-AllowReconnect"`
27.7.1.3 AsynchUpdates

**Description**  This option specifies that the Notification Service should send subscription updates to suppliers and publication (offer) updates to consumers asynchronously. If this option is used, a separate thread will be allocated for dispatching these subscription/publication updates.

**Usage**  Use the `-AsynchUpdates` option to enable asynchronous sending of subscription and publication updates. The number of threads used for sending these updates defaults to one and is not configurable.

**Impact**  When subscription or publication changes occur, the Notification Service dispatches updates to registered listeners. By default, these updates are performed on the same thread as that on which the subscription or publication changes occur. Using a separate thread for sending these updates decouples consumers making subscription changes from suppliers receiving subscription updates, and decouples suppliers making publication changes from consumers receiving publication updates. Using a separate thread for sending subscription or publication updates can improve performance. The allocation of a separate thread for these updates consumes additional resources.

**See Also**  27.4.4, 27.7.1.11

**Example**  static TAO_CosNotify_Service "-AsynchUpdates"
27.7 Notification Service Configuration Options

27.7.1.4 DispatchingThreads \textit{nthreads}

\textbf{Description} This option specifies the number of threads to be created and used for multithreaded dispatching of events.

\textbf{Usage} Using this option gives you the benefit of separating the dispatching thread from the processing thread. This separation allows for greater decoupling between suppliers and consumers. If this option is not used, then the same thread used to process the event is also used to dispatch it to the consumers. By default, the number of threads specified for this option are allocated as a pool for each consumer admin object. When used with the \texttt{-AllocatedTaskPerProxy} option, the specified number of threads is allocated per supplier proxy, which is effectively per consumer.

\textbf{Impact} Specifying additional dispatching threads results in increased decoupling between consumers and suppliers. It may also increase throughput for heavy loads. However, additional dispatching threads consume more resources and may introduce higher event latencies due to context switches.

\textbf{See Also} 27.7.1.1

\textbf{Example} \texttt{static TAO_CosNotify_Service "-DispatchingThreads 5"}
27.7.1.5 **ListenerThreads nthreads [DEPRECATED]**

**Description**
This option is deprecated.

This option specifies the number of threads to be created and used for listener (proxy supplier) filter evaluation.

**Usage**
This activity now occurs on the dispatching threads and the threads specified by this option are now simply treated as additional dispatching threads. You should no longer use this option and should simply define your total desired dispatching threads directly using `-DispatchingThreads`.

**Impact**
See 27.7.1.4 for dispatching threads impact.

**See Also**
27.7.1.4

**Example**
static TAO_CosNotify_Service "-ListenerThreads 5"
27.7 Notification Service Configuration Options

27.7.1.6 LookupThreads nthreads [DEPRECATED]

**Description**
This option is deprecated.

This option specifies the number of threads to be created and used for subscription lookup.

**Usage**
This activity now occurs on the source thread. If -AllocateTaskPerProxy is specified, then this option is ignored. If -AllocateTaskPerProxy is not specified, then the threads specified by this option are simply treated as additional source processing threads. You should no longer use this option and should simply define your total desired source processing threads directly using -SourceThreads.

**Impact**
See 27.7.1.12 for source processing threads impact

**See Also**
27.7.1.12

**Example**
static TAO_CosNotify_Service "-LookupThreads 5"
27.7.1.7 MTDispatching [DEPRECATED]

Description: This option is deprecated.

This option was previously required to force the notification channel to use a separate dispatching thread pool. This functionality is now invoked implicitly when using the -DispatchingThreads option.

Usage: Use the -DispatchingThreads option to specify the number of dispatching threads and to force multithreaded dispatching.

Impact: See 27.7.1.4 for dispatching threads impact.

See Also: 27.7.1.4

Example:

This example forces multithreaded dispatching with a pool of one dispatching thread.

```cpp
static TAO_CosNotify_Service "-DispatchingThreads 1"
```
27.7 Notification Service Configuration Options

27.7.1.8 MTListenerEval [DEPRECATED]

Description  This option is deprecated.

This option was previously required to force the notification channel to use a separate listener evaluation thread pool. This functionality is no longer allocated its own thread pool.

Usage  Use the -SourceThreads and -DispatchingThreads options to allocate the desired thread pools.

Impact  N/A

See Also  27.7.1.4, 27.7.1.12

Example  N/A
27.7.1.9 MTLookup [DEPRECATED]

Description  This option is deprecated.

This option was previously required to force the notification channel to use a separate listener evaluation thread pool. This functionality is no longer allocated its own thread pool.

Usage  Use the -SourceThreads and -DispatchingThreads options to allocate the desired thread pools.

Impact  N/A

See Also  27.7.1.4, 27.7.1.12

Example  N/A
27.7 Notification Service Configuration Options

27.7.1.10 MTSourceEval [DEPRECATED]

Description  This option is deprecated.

This option was previously required to force the notification channel to use a separate source evaluation thread pool. This functionality is now invoked implicitly when using the -SourceThreads option.

Usage  Use the -SourceThreads option to specify the number of source evaluation threads and to force multithreaded processing.

Impact  See 27.7.1.12 for dispatching threads impact.

See Also  27.7.1.12

Example  static TAO_CosNotify_Service "-SourceThreads"
27.7.1.11 NoUpdates

Description This option globally disables subscription and publication updates.

Usage When specified, this option causes all subscription and publication updates to be disabled. Any consumers or suppliers subscribing to these updates will never receive any.

Impact Consumers and suppliers that depend on subscription and publication updates may not function correctly. There should be a small decrease in processor resource usage when this option is enabled.

See Also 27.7.1.3

Example static TAO_CosNotify_Service "-NoUpdates"
27.7 Notification Service Configuration Options

27.7.1.12 SourceThreads nthreads

Description This option specifies the number of threads to be created and used for multithreaded source processing of events.

Usage Using this option gives you the benefit of separating the source processing thread from the ORB’s thread that is used to process an incoming push() operation. This separation allows for greater decoupling between suppliers and consumers. If this option is not used, then the same thread used to process the incoming event is also used for consumer proxy filter evaluation and fan out to the receiving supplier proxies. By default, the number of threads specified for this option are allocated as a pool for each supplier admin object. When used with the -AllocatedTaskPerProxy option, the specified number of threads is allocated per consumer proxy, which is effectively per supplier.

Impact Specifying additional source processing threads results in increased decoupling between consumers and suppliers. It may also increase throughput for heavy loads. However, additional source processing threads consume more resources and may introduce higher event latencies due to context switches.

See Also 27.7.1.1

Example static TAO_CosNotify_Service "-SourceThreads 5"
27.7.2 Connection Reliability: Topology Persistence Options

To support setting the QoS property ConnectionReliability to Persistent (as discussed in 27.3.6), we must specify the Topology_Factory that the notification channel uses. The only topology factory currently supported is the XML topology factory which is located in the TAO_CosNotification_Persist library. We typically load this library and factory using a dynamic directive in the service configurator file. Here is the general form of this directive (which should all appear on one line of the configuration file):

dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist: make_XML_Topology_Factory() "options"

This dynamically loads the XML Topology Factory as the Topology_Factory and configures it with the specified options. The rest of this section documents the various options that this factory allows.

Application developers are also free to write their own topology factories, but that is left as an exercise to the reader.

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v</td>
<td>27.7.2.6</td>
<td>Enable verbose logging for topology factory configuration.</td>
</tr>
<tr>
<td>-base_path</td>
<td>27.7.2.2</td>
<td>Specifies base path to use for saving and loading topology files.</td>
</tr>
<tr>
<td>-save_base_path</td>
<td>27.7.2.5</td>
<td>Specifies a specific base path to use for saving topology files.</td>
</tr>
<tr>
<td>-load_base_path</td>
<td>27.7.2.3</td>
<td>Specifies a specific base path to use for loading topology files.</td>
</tr>
<tr>
<td>-backup_count</td>
<td>27.7.2.1</td>
<td>Specifies the number of topology backup files to keep.</td>
</tr>
<tr>
<td>-no_timestamp</td>
<td>27.7.2.4</td>
<td>Suppresses the writing of a time stamp in the topology file.</td>
</tr>
</tbody>
</table>
27.7.2.1 **backup_count**

**Description**  This option specifies how many previous versions of the XML file will be retained.

**Usage**  This option is not required. The default value is 1.

The default value, 1, means that only the `file_path.000` file will be kept. If a higher number is specified, then older versions will be kept. The older versions will be named `file_path.001`, `file_path.002`, and so on.

**Impact**  Under normal circumstances, only one backup file is required; setting this number to a larger value lets the system keep a brief history of topology changes. Since the XML files are roughly human-readable, this can be used as a diagnostic tool for problems related to Notification Service topology persistence.

**See Also**  27.7.2.2

**Example**  ```
dynamic Topology Factory Service Object*
TAO_CosNotification_Persist: make_XML_Topology_Factory() "-base_path
/safe/topology -backup_count 2"
```
27.7.2.2  **base_path** `file_path`

**Description**  This option determines where persistent topology information is loaded from and stored.

**Usage**  The argument for this option is a fully qualified path name without an extension. Three extensions will be appended to this path to create three file names: `.new`, `.xml`, and `.000`.

This option is not required. Its default value is `.Notification_Service_Topology`.

**Impact**  Saved topology information will be written to `file_path.new` file. Information in a topology persistence file with a `.new` extension is not necessarily complete and will not be used to restore the topology.

When the `file_path.new` file is complete, the previous `file_path.000` file (if any) will be deleted, the previous `file_path.xml` file (if any) will be renamed to `file_path.000` and the `file_path.new` file will be renamed to `file_path.xml`.

The assumption is that a file system rename operation is atomic. If this assumption holds, then at any time the file `file_path.xml` (if it exists) contains the most recent complete save. If `file_path.xml` does not exist, then `file_path.000` contains the most recent complete save. If neither of these files exist, the saved topology information is not available.

**See Also**  27.7.2.1, 27.7.2.5, 27.7.2.3

**Example**  
```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:make_XML_Topology_Factory() "-base_path
/safe/topology"
```
27.7 Notification Service Configuration Options

27.7.2.3 load_base_path file_path

Description This option specifies the base path to be used for the files that load the topology state. When used with the -store_base_path option they provide an alternative to the -base_path option. They allow the file from which topology information is loaded at Notification Service startup time to be different from the file to which this information is saved as the system runs.

Usage This option is not required. There is no default value.
This option should not be used if the -base_path option is used.
If this option is used the -save_base_path option should also be used.

Impact This option is mostly used for developer testing. A system administrator may find an interesting use for this option, possibly involving script files that rename the XML files during recovery from a Notification Service failure.

See Also 27.7.2.2, 27.7.2.5

Example
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist: make_XML_Topo loy_Factory() "-load_base_path /safe/topology_in -save_base_path /safe/topology_out"
27.7.2.4 no_timestamp

**Description**  
This option suppresses the time stamp that is normally written to the XML file in which the topology is saved.

**Usage**  
Use this option when you don’t want the timestamp written such as when you wish to automate comparisons of XML topology persistence files.

**Impact**  
The time stamp is for information only and is not needed for correct functioning of the topology persistence.

This option is intended primarily for testing the persistent topology implementation.

**Example**  
dynamic Topology.Factory Service_Object*  
TAO_CosNotification_Persist: _make_XML_Topology_Factory() "-no_timestamp"
27.7 Notification Service Configuration Options

27.7.2.5 **save_base_path** file_path

**Description** This option specifies the base path to be used for the files that save the topology state. When used with the -load_base_path option they provide an alternative to the -base_path option. They allow the file from which topology information is loaded at Notification Service startup time to be different from the file to which this information is saved as the system runs.

**Usage** This option is not required. There is no default value.

This option should not be used if the -base_path option is used.

If this option is used the -load_base_path option should also be used.

**Impact** This option is mostly used for developer testing. A system administrator may find an interesting use for this option, possibly involving script files that rename the XML files during recovery from a Notification Service failure.

**See Also** 27.7.2.2, 27.7.2.3

**Example**
```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist: make_XML_Topology_Factory() "-load_base_path /safe/topology_in -save_base_path /safe/topology_out"
```
27.7.2.6  v

**Description**  This option enables verbose logging. This option and any option that appears after it will be written to the log (normally the console) as it is processed. This is intended to help diagnose and document the Topology Persistence settings. This output is also enabled when the TAO debug level is greater than zero.

The default is to configure Topology Persistence silently.

**Usage**  This should be the first option for the Topology Factory in the service configurator file so that all options will be displayed.

**Example**  
```
dynamic Topology.Factory Service_Object*
TAO_CosNotification_Persist::make_XML_Topology.Factory() "-v"
```
27.7 Notification Service Configuration Options

27.7.3 Event Reliability: Event Persistence Options

To support setting the QoS property `EventReliability` to `Persistent` (as discussed in 27.3.7), we must specify the `Event_Persistence` factory that the notification channel uses. The only event persistence factory currently supported is the standard event persistence factory and we typically load this factory using a dynamic directive in the service configurator file. Here is the general form of this directive (which should all appear on one line of the configuration file):

```
dynamic Event_Persistence Service_Object*
TAO_CosNotification:_make_Standard_Event_Persistence() "options"
```

This dynamically loads the standard event persistence factory as the `Event_Persistence` factory and configures it with the specified options. The rest of this section documents the various options that this factory allows. Application developers are also free to write their own event persistence factories, but that is left as an exercise to the reader.

<table>
<thead>
<tr>
<th>Option</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-v</code></td>
<td>27.7.3.3</td>
<td>Enable verbose logging for event persistence factory configuration.</td>
</tr>
<tr>
<td><code>-file_path</code></td>
<td>27.7.3.2</td>
<td>Specifies the file to use for saving and loading event persistence.</td>
</tr>
<tr>
<td><code>-block_size</code></td>
<td>27.7.3.1</td>
<td>Specifies a specific base path to use for saving topology files.</td>
</tr>
</tbody>
</table>
27.7.3.1  block_size

Description  This option gives the block size in bytes for the device on which the event reliability file is stored.

Usage  This option is not required. The default value is 512.

Impact  For both performance and reliability reasons, it is important that the value matches the physical characteristics of the device.

For Windows and for “normal” drives (IDE drives and most SCSI drives) on Linux and UNIX, the correct value is almost always 512.

For special purpose SCSI drives or devices in a RAID array, the correct block size is determined by the values used to format the disk and configure the system. See information provided by your vendor and/or system administrator.

Example  dynamic Event_Persistence_Service_Object*
TAO_CosNotification:_make_Standard_Event_Persistence() "-file_path /safe/events -block_size 1024"
27.7 Notification Service Configuration Options

27.7.3.2 file_path

Description  This option gives the completely qualified name for the file in which persistent event information will be stored.

This is a required option. There is no default value.

Impact  The file should be configured on a reliable device that supports synchronized writes (i.e., flushing the operating system’s write cache). A device that is suitable for storing a reliable database would be appropriate for storing this file. The file will be subject to a relatively high number of small (single block) write requests, but very few, if any, read requests.

If the file does not exist, a new file will be created.

If the file does exist, and if topology is successfully loaded, the events from this file will be reloaded and redelivered automatically.

Example  

dynamic Event_Persistence Service_Object*
TAO_CosNotification:make_Standard_Event_Persistence() "-file_path /safe/events"
**27.7.3.3 v**

**Description**
This option enables verbose logging. This option and any option that appears after it will be written to the log (normally the console) as it is processed. This is intended to help diagnose and document the Event Persistence settings. This output is also enabled when the TAO debug level is greater than zero.

The default is to configure Event Persistence silently.

**Usage**
This should be the first option for the Event_Persistence factory in the service configurator file so that all options will be displayed.

**Example**
dynamic Event_Persistence Service_Object*
TAO_CosNotification: make_Standard_Event_Persistence() "-v -file_path events"
CHAPTER 28

Interface Repository

28.1 Introduction

The Interface Repository (IFR) is a CORBA service defined by the OMG in Chapter 10 of the CORBA 3.0.3 specification (OMG Document formal/04-03-12). The IFR manages information related to IDL entities. It is able to maintain relationships between such things as modules, interfaces, and operations, as well as all of the supporting entities, such as typedefs, primitives, and structs. The IFR has its own set of interfaces, used to store, manage, and retrieve information in the repository. See 10.5 and 10.7 of CORBA 3.0.3 for a full description of the IDL defining the IFR.

Why is it important to have a repository of information about IDL interfaces and types? It gives CORBA objects the means for self description. It is similar to Java’s Reflection API. Through the IFR, it is possible to obtain an otherwise unknown object reference, and to be able to make invocations on it. This is useful when building applications such as system management or debugging tools, middleware gateways or protocol translators. It is also useful when used in conjunction with, for instance, the Trader service. For example, you may have a number of Objects which offer a “print” capability to deliver
data to a printer, each one tuned to a specific kind of job. You may not wish to compile in a number of similar stubs since your application may be older than the objects with which it wishes to interact. Using the IFR, it is possible to determine which object has the appropriate interface.

28.2 Using the Interface Repository

The IFR maintains an object model representing the contents of interfaces and related IDL components. The IFR may hold the contents of many IDL files, all anchored by a single CORBA::Repository object. The repository has an interface that allows for addition of modules, interfaces, exceptions, typedefs, anything that can legally exist outside the context of a module in IDL. The Repository also has a query interface allowing for retrieval of this information. See 28.4 for an example use of this query interface. The full details of the IFR’s storage and retrieval interface are beyond the scope of this book (see 10.5 and 10.7 in the CORBA specification).

In addition to the repository, there are interfaces that describe all of the IDL constructs. For instance, ModuleDef is used to describe a module, InterfaceDef for interfaces. These definition interfaces are all part of the CORBA domain, and they all inherit from either Container, Contained, or both.

The definition of any entity which may contain another entity, such as a ModuleDef, or InterfaceDef, derives from Container (for the sake of brevity, since all of these types are defined in the CORBA module, the prefix "CORBA:::" will be omitted). A Container has an interface which allows searching over the contents of the container, with control over the scope of the search, and the level of detail returned.

The definition of any entity which may be contained by another, which includes all but the repository itself, derives from Contained. These interfaces all inherit a common set of attributes including a reference to the enclosing container, a Repository ID string uniquely identifying this entity, and the name given to the entity in the original IDL.

The CORBA specification describes the Interface Repository as a tool for the “management of a collection of related objects’ interface definitions.” It does not require, however, that all the IR Objects in a repository be “related.” If they are unrelated, or if a repository is used as a long-running catch-all, an
28.3 TAO’s Interface Repository Implementation

TAO’s implementation of the Interface Repository consists of two processes. The first, IFR_Service, is meant to be always available. It houses the Repository objects and responds to queries. The other is the IFR loader, tao_ifr, a utility used to interpret IDL files and load the results into the IFR. It also provides other useful maintenance services. The source for both of these applications is located in $TAO_ROOT/orbsvcs/IFR_Service.

28.3.1 IFR_Service

This is the application supplied with TAO that is host to Repository objects. The IFR_Service application gives a measure of control over the repository. As with any TAO application, IFR_Service is subject to all of the ORB configuration options documented in Chapter 19. Table 28-1 lists the additional command line options that are specific to IFR_Service.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o filename</td>
<td>Specify a filename for storing the IOR from this server.</td>
<td>IOR is written to if_repo.ior</td>
</tr>
<tr>
<td>-p</td>
<td>Make IFR service persistent using a memory-mapped file.</td>
<td>IFR does not use memory-mapped files.</td>
</tr>
<tr>
<td>-b filename</td>
<td>Specify a filename to use as backing store with -p option.</td>
<td>ifr_default_backing_store is used for the mapping file,</td>
</tr>
</tbody>
</table>
When started, IFR_Service always writes the Repository IOR to a file. By default, the file named if_repo.ior is used. The name of this file may be changed by using the -o option. Clients may also locate the IFR service via IP multicast. When multicast support is enabled with -m 1, the default port on which the Interface Repository listens for service discovery multicast requests is 10020. This is defined in $TAO_ROOT/tao/default_ports.h. The environment variable InterfaceRepoServicePort can be used to override this default. In addition, the clients may also locate the service by placing a stringified object reference in the InterfaceRepositoryIOR environment variable.

By default, the repository’s state is maintained in the process’s heap. It is frequently desirable to use persistent storage, allowing the Repository to withstand a process termination and not be populated each time it is started. The -p option is used to enable the use of a memory-mapped file for the repository’s state. The name of the backing store file may be set using the -b option. When run on a Win32 based platform, the IFR_Service may use the windows registry as its database, when started with -r. Use of the memory-mapped file takes precedence over the registry. In other words, -p takes precedence over -r.

The final command line option to discuss is the -l option, which activates the use of mutex locks in a multithreaded IFR_Service. This is recommended when multiple updates may occur concurrently.

### Table 28-1 Interface Repository Service Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-l</td>
<td>Lock repository during operation invocations.</td>
<td>No locking is used.</td>
</tr>
<tr>
<td>-r</td>
<td>Use the Windows registry as backing store, if available. Overridden by -p.</td>
<td>Do not use the windows registry as a backing store.</td>
</tr>
<tr>
<td>-m 0</td>
<td>Specify whether the IFR Service should listen for multicast service discovery requests. If -m 1 is used, it listens for multicast requests.</td>
<td>Do not listen for multicast service location requests.</td>
</tr>
</tbody>
</table>
28.3 TAO’s Interface Repository Implementation

28.3.2 tao_ifr

The tao_ifr executable is TAO’s interface repository loader and makes the task of populating the IFR as simple as possible. It works by reading IDL files, supplied as command line arguments, then adding (or removing) the entities contained within. It uses the same front end and structural code as the TAO IDL compiler, tao_idl. The front end is responsible for parsing and processing the IDL files into internal data structures. The differences between the IFR loader and IDL compiler lie in their back ends. The tao_ifr back end populates the Interface Repository server with the IDL information it processes. Like the IDL compiler, the IFR loader is applied to one or more IDL files. The general form of the command line for tao_ifr is:

```
tao_ifr [-ORB options][ifr options] idl_file [idl_file ...]
```

See Chapter 19 for information on the ORB initialization options, all of which are available here. Because they share a common front end, options intended for this code are identical to those of the TAO IDL compiler. Table 28-2 describes all of front end-related command line options that the IFR loader shares with the IDL compiler.

Table 28-2 tao_ifr Front End Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v</td>
<td>Traces tao_ifr processing stages.</td>
</tr>
<tr>
<td>-V</td>
<td>Prints the version info for tao_ifr and exits.</td>
</tr>
<tr>
<td>-d</td>
<td>Outputs a dump of the Abstract Symbol Tree (AST).</td>
</tr>
<tr>
<td>-u</td>
<td>Prints the list of options and exits.</td>
</tr>
<tr>
<td>-E</td>
<td>Invokes the preprocessor and exits.</td>
</tr>
<tr>
<td>-D macro_def</td>
<td>Preprocessor defines macro_def</td>
</tr>
<tr>
<td>-U macro_name</td>
<td>Preprocessor undefines macro_name.</td>
</tr>
<tr>
<td>-I include_path</td>
<td>include_path is passed to the preprocessor.</td>
</tr>
<tr>
<td>-A assertion</td>
<td>assertion (a local implementation-specific escape) is passed to the preprocessor.</td>
</tr>
<tr>
<td>-Y path</td>
<td>Specifies a path to the preprocessor, rather than what was used to build tao_ifr.</td>
</tr>
<tr>
<td>-t dir_name</td>
<td>Temporary directory to be used by the tao_ifr front end. If not specified, then ACE_DEFAULT_TEMP_DIR_ENV or /tmp is used.</td>
</tr>
</tbody>
</table>
These command line options can be broken into two groups. The first group is essentially for debugging: these are \(-v\), \(-V\), and \(-d\). The remainder control the processing of the IDL files. All of the options above are identical to those of the IDL compiler.

Table 28-3 describes all of command line options that are unique to the interface loader.

**Table 28-3 tao_ifr Back End Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-Si)</td>
<td>Suppress processing of included IDL files.</td>
</tr>
<tr>
<td>(-L)</td>
<td>Enable locking at IDL file level.</td>
</tr>
<tr>
<td>(-r)</td>
<td>Contents of IDL file(s) are removed from, not added to, the repository.</td>
</tr>
</tbody>
</table>

By default, the IFR loader processes all IDL found in the files specified to it and all their included files and adds this information to the repository. The \(-Si\) option cause the IFR loader to only add information from IDL files specified on the command line and not from included files. The \(-r\) option causes all processed IDL elements to be removed from the IFR repository.

The TAO IFR loader uses the same techniques for obtaining a reference to the Repository as any other CORBA process. It asks the ORB to resolve the initial reference of "InterfaceRepository." The ORB will try an initial reference supplied on the command line (see 19.8.13). Use "InterfaceRepository" for the Object ID, and use "file://if_repo.ior" for the Object URL. Of course, if IFR_Service was started with \(-o filename\), then use that file name instead. Otherwise, the ORB may locate the IOR via the InterfaceRepositoryIOR environment variable or an IP multicast request (assuming the IFR server was started with multicast enabled).
28.4 Example IFR Client

Now we take a look at how an application can use the IFR to obtain information about an interface. This program, called IFRBrowser, obtains a reference to the Repository, then traverses all of the entities, writing them to the console. Full source code for this example is in the TAO 1.4a source code distribution in the directory

$TAO_ROOT/DevGuideExamples/InterfaceRepo/.

28.4.1 Annotated Source Code

The following code is from IFRBrowser.cpp.

The CORBA specification does not define a location for the IFR interface headers. All of the interfaces this browser uses are defined in IFR_Basic.idl, found in $TAO_ROOT/tao/IFR_Client. Therefore this browser needs the client side include file shown below. The Log_Msg.h include file provides access to the ACE logging facility that provides the ACE_DEBUG macro.

```c++
#include <ace/Log_Msg.h>
#include <tao/IFR_Client/IFR_BasicC.h>

const char* programLabel = "IFR Browser";
```

The following forward declarations are for methods used in main(), but defined further below. They represent the methods used to output certain elements of the IFR.

```c++
void listContents(const CORBA::ContainedSeq& repoContents);
void listInterface(CORBA::InterfaceDef_ptr interfaceDef);
void listOperation(CORBA::OperationDescription* operationDescr);
void listParameter(CORBA::ParameterDescription* parameterDescr);

const char* decodeTypeCode(const CORBA::TypeCode_ptr typeCode);
const char* decodeParameterMode(CORBA::ParameterMode mode);
const char* decodeOperationMode(CORBA::OperationMode mode);
```

The main function performs all of the common CORBA application initialization, obtains a reference to the Interface Repository, then lets the listRepo() function handle the actual work.

```c++
int main(int argc, char* argv[])
```
{ try
  CORBA::ORB_var orb = 
  CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj =
  orb->resolve_initial_references("InterfaceRepository");
  CORBA::Repository_var ifrRepo = 
  CORBA::Repository::_narrow(obj.in());
  if(CORBA::is_nil(ifrRepo.in())) {
    ACE_DEBUG((LM_ERROR, 
      ACE_TEXT("(%N) failed to narrow interface repository reference.
")));
    return -1;
  }

  The Repository may hold many elements. The operation contents() retrieves all of these repository-level elements in a single sequence. The description kind value CORBA::dk_all indicates that all elements are to be included in the list. The kind value may be used to constrain the list to only particular kinds of elements, such as typedefs or interfaces.

  CORBA::ContainedSeq_var repoContents =
  ifrRepo->contents(CORBA::dk_all,1);

  ACE_DEBUG((LM_INFO, 
    ACE_TEXT("%s: the interface repository contains %d elements.\n"), 
    programLabel, 
    repoContents.length())
  ));
  listContents(repoContents.in());
  return 0;
}
catch(CORBA::Exception& ex) { 
  ex._tao_print_exception("Main block"); 
  return 1;
}

The listContents() function iterates over the sequence of contained objects, some of which may in turn contain other objects. The Repository may directly contain anything that can be defined outside the context of a module, which traditionally includes constants, type definitions, exceptions, interfaces,
valuetypes, value boxes, and modules. With the addition of the CORBA Component Model (CCM) this can also include things such as components and homes. This example only supports constants, typedefs, exceptions, interfaces, and modules. It ignores all other constructs.

```c
void listContents(const CORBA::ContainedSeq& repoContents)
{
    // List the contents of each element.
    //
    for(unsigned int i = 0; i < repoContents.length(); ++i) {
        switch(((repoContents[i])->describe())->kind) {
            case CORBA::dk_Constant:
                ACE_DEBUG((LM_INFO,
                    ACE_TEXT("%s: element[%d] is a constant definition.\n"),
                    programLabel,
                    i + 1
                ));
                break;
            case CORBA::dk_Typedef:
                ACE_DEBUG((LM_INFO,
                    ACE_TEXT("%s: element[%d] is a typedef definition.\n"),
                    programLabel,
                    i + 1
                ));
                break;
            case CORBA::dk_Exception:
                ACE_DEBUG((LM_INFO,
                    ACE_TEXT("%s: element[%d] is an exception definition.\n"),
                    programLabel,
                    i + 1
                ));
                break;
            case CORBA::dk_Interface:
                {
                    ACE_DEBUG((LM_INFO,
                        ACE_TEXT("%s: element[%d] is an interface definition.\n")
                        ACE_TEXT("%s: listing element[%d]...\n"),
                        programLabel,
                        i + 1,
                        programLabel,
                        i + 1
                    ));
                }
                break;
        }
    }

    When the kind of element encountered is an interface, IFRBrowser narrows the contained reference to an InterfaceDef, which is the definition of an
interface. Then, `listInterface()` is used to output the particulars for the interface.

```cpp
CORBA::InterfaceDef_var interfaceDef =
    CORBA::InterfaceDef::narrow(repoContents[i].in());
listInterface(interfaceDef.in());
break;
}
```

As with the interface description, a module may also contain other elements. No special function is needed for a module as the interface `ModuleDef` is nothing more than the empty inheritance of `Container` and `Contained`. The `Contained` interface has all the necessary identity information access.

```cpp
CORBA::ModuleDef_var moduleDef =
    CORBA::ModuleDef::narrow(repoContents[i].in());
CORBA::ContainedSeq_var moduleContents =
    moduleDef->contents(CORBA::dk_all,1);
CORBA::String_var moduleId = moduleDef->id();
CORBA::String_var moduleName = moduleDef->name();
ACE_DEBUG((LM_INFO,
    ACE_TEXT("%s: \
      module %s
    {
")
    programLabel,
    moduleId.in(),
    moduleName.in(),
    repoContents.length(),
    programLabel,
    moduleContents.in()));
```

When an interface is encountered above, `listInterface()` is used to output all of the particulars. Note below, that a specialized interface is used to obtain
the identity information. This differs from the ModuleDef, which relies on the inherited Contained interface for identity. Since InterfaceDef also inherits from Contained, the name and repository Id could be accessed through the InterfaceDef directly. The FullInterfaceDescription provides access to the basic identity as well as to other interface-specific information, thus providing “one stop shopping” for all of the interface’s details.

```c
void listInterface(CORBA::InterfaceDef_ptr interfaceDef)
{
    CORBA::InterfaceDef::FullInterfaceDescription_var fullDescr =
        interfaceDef->describe_interface();

    const char* interfaceName =
        fullDescr->name;
    const char* interfaceRepoId =
        fullDescr->id;

    ACE_DEBUG((LM_INFO,
        ACE_TEXT("%s:\n	// %s
	interface %s
	{"),
        programLabel,
        interfaceRepoId,
        interfaceName
    ));

    unsigned int operationsCount;
    if((operationsCount =
         fullDescr->operations.length())
        > 0
    )
    {
        for(unsigned int i = 0; i < operationsCount; ++i) {
            listOperation(&fullDescr->operations[i]);
        }
    }

    unsigned int attributesCount;
    if((attributesCount =
         fullDescr->attributes.length())
        > 0
    )
    {
        ACE_DEBUG((LM_INFO,
            ACE_TEXT("%s: %s has %d attribute(s).\n"),
            programLabel,
            interfaceName,
            attributesCount
        ));
    }
}
For each operation in an interface, the listOperation() method lists the repository Id, name, return type, and all of the parameters.

```c
void listOperation(CORBA::OperationDescription* operationDescr)
{
    const char* operationName =
        operationDescr->name;
    const char* operationRepoId =
        operationDescr->id;
    const char* operationResult =
        decodeTypeCode(operationDescr->result.in());
    const char* operationMode =
        decodeOperationMode(operationDescr->mode);

    ACE_DEBUG((LM_INFO,
        ACE_TEXT("\n\t	// %s \n\t	%s %s %s"),
        operationRepoId,
        operationResult,
        operationName,
        operationMode
    ));

    CORBA::ParDescriptionSeq* params =
        &(operationDescr->parameters);
    CORBA::ULong paramsCount = params->length();
    if (paramsCount > 0) {
        ACE_DEBUG((LM_INFO, "\n\t	(\n\t	"));
        for (CORBA::ULong i =0; i < paramsCount; ++i) {
            listParameter(&(*params)[i]);
            if(i < (paramsCount - 1)) {
                ACE_DEBUG((LM_INFO, ",\n\t	"));
            }
        }
        ACE_DEBUG((LM_INFO, ";\n\t	)\n"));
    } else {
        ACE_DEBUG((LM_INFO, "()\n"));
    }
}
```

Finally, each parameter to an operation is listed, one per line, based on the contents of the supplied parameter description. This information could also be
used to populate an argument list on a dynamic request object, if DII were used to interact with an object implementing this interface.

```c
void listParameter(CORBA::ParameterDescription* parameterDescr)
{
    const char* typCode =
        decodeTypeCode(parameterDescr->type.in());
    const char* paramMode =
        decodeParameterMode(parameterDescr->mode);
    ACE_DEBUG((LM_INFO,
        ACE_TEXT("%s %s %s",
        paramMode,
        typCode,
        parameterDescr->name.in()));
}
```

The remaining helper functions are used to return strings based on various codes.

```c
const char* decodeTypeCode(const CORBA::TypeCode_ptr typeCode)
{
    const char* code;
    if(typeCode->equivalent(CORBA::_tc_void))
        code = "void";
    else if(typeCode->equivalent(CORBA::_tc_boolean))
        code = "boolean";
    else if(typeCode->equivalent(CORBA::_tc_string))
        code = "string";
    else
        code = "";
    return code;
}

const char* decodeParameterMode(CORBA::ParameterMode mode)
{
    const char* paramMode;
    switch(mode) {
    case CORBA::PARAM_IN:
        paramMode = "in";
        break;
    case CORBA::PARAM_OUT:
        paramMode = "out";
        break;
    case CORBA::PARAM_INOUT:
        paramMode = "inout";
        break;
    default:
```
Interface Repository

```cpp
paramMode = ""
}
return paramMode;
}

const char* decodeOperationMode(CORBA::OperationMode mode)
{
  return (mode == CORBA::OP_NORMAL) ? "synchronous" : "asynchronous";
}
```

### 28.4.2 Run the Example

For this example, we have a simple IDL file, `test.idl`. The implementation of the IDL is not important to this example. However, this IDL file shows several major components of IDL, including a module, an interface and an operation.

```idl
module warehouse
{
  interface inventory
  {
    boolean getCDinfo (in string artist,
                       inout string title,
                       out float price);
  };
};
```

The following commands show how to start the Interface Repository service, and how to populate the repository with the elements of the IDL file shown above.

```
IFR_Service -m 1&
tao_ifr test.idl
```

The first command starts the `IFR_Service` with multicast discovery enabled. We will rely on clients using multicast to locate the Interface Repository. The `IFR_Service` is normally a long-running process, so this example runs it in the background (assuming some Unix command shell). When first run, the Repository is empty. The `tao_ifr` command above reads the contents of `test.idl` and populates the Repository. At this point, we run the `IFRBrowser`.

```
IFRBrowser
```
IFRBrowser uses multicast to locate the Repository, then iterates over the entire contents of the Repository, writing the results as shown.

IFR Browser: the interface repository contains 1 elements
IFR Browser: element[1] is a module definition.
IFR Browser:
   // IDL:warehouse:1.0
module warehouse
{
  IFR Browser: the module contains 1 elements.
  IFR Browser: element[1] is an interface definition.
  IFR Browser: listing element[1]...
     // IDL:warehouse/inventory:1.0
     interface inventory
     {
       // IDL:warehouse/inventory/getCDinfo:1.0
       boolean getCDinfo synchronous
       {
         in string artist,
         inout string title,
         out price
       };
     }
}

Alternatively, we could run this example using the InterfaceRepositoryIOR environment variable. The following code assumes the use of bash (or another similar shell environment):

IFR_Service -o ifr.ior&
export InterfaceRepositoryIOR=file://ifr.ior
tao_ifr test.idl
IFRBrowser
Preface

Security has changed in TAO 1.4a. In this release, the version of TAO from the DOC Group that forms the basis for OCI’s TAO 1.4a is in a state of transition. Previous versions of TAO included a partial implementation of CORBA Security and SSLIOP. The implementation in TAO 1.4a is based on newer versions of the CORBA Security specification. Unfortunately, the implementation is incomplete, so some features are not available. Further, some features that were available in previous releases may not be available as the implementation currently exists. Specifically, Policy Enforcing applications (see 29.8, “Security Policy Enforcing Application”) will very likely not work properly, though all Security Unaware (see 29.6, “Security Unaware Application”) and many Policy Changing (see 29.7, “Security Policy Controlling Application”) applications will work. The rest of this chapter documents these working (or mostly working) features.

Why is CORBA Security changing? In the original specification for the CORBA Security Service (OMG Document security/00-12-02), the OMG defined a model for security by specifying sets of features required for
security. These feature sets were further divided into feature packages (see Table 29-1). The specification also defined IDL interfaces (effectively, APIs) for accessing these various feature sets from within applications.

One of the more critical feature packages was “Common Secure Interoperability,” or “CSI,” which was supposed to allow different vendors’ implementations of security to interoperate, so that a secure CORBA application need not be deployed in an ORB-homogenous environment. Unfortunately, as implementations for CSI emerged and evolved (one of these implementations was in TAO), developers recognized that the original CSI package merely addressed the transport layer and was inadequate for truly interoperable security systems. Thus, the OMG augmented this feature package with CSIv2 (Chapter 24 of CORBA 3.0.3, or Chapter 26 of CORBA/IIOP 2.6), which layers atop CSIv1 and communicates privileges under which requests are made.

As developers progressed on implementing CSIv2, they also realized that the existing APIs (IDL modules SecurityLevel1 and SecurityLevel2) were insufficient to use with CSIv2. Thus, as one of the principal members of the CSIv2 task force developed a Java implementation of CORBA Security, he concurrently created SecurityLevel3, an API which, in the developer’s words, is “much more robust...than OMG’s Security Level 2 API, and has well defined semantics.” We anticipate that SecurityLevel3 will eventually be pushed into an official OMG specification. Also, recognizing the deficiencies of the Security Levels 1 and 2 APIs, developers of other security implementations for other ORBs began implementing CSIv2 under Security Level 3 APIs. TAO contains one of those implementations.

Unfortunately, the TAO 1.4a release comes at a point in the DOC Group release schedule that is between these implementations. The newer SecurityLevel3 module replaces SecurityLevel1 and SecurityLevel2, and there is no design for coexistence. Although many in the TAO community seem to be interested in using security features, few have shown interest in sponsoring research or development activity in that area. That, of course, means that development moves forward on a time-permitting basis from volunteers who already have other sponsored work.

TAO’s security implementation will continue to grow and improve, hopefully accelerated by active sponsorship from the community-at-large.
29.1 Introduction

This chapter introduces you to various features of TAO’s Security Service:

- The OMG’s Security Service Specification.
- Modern cryptographic techniques.
- The Secure Sockets Layer (SSL) protocol.
- TAO’s fundamental capabilities with respect to distributed object system security.

Since we cannot provide a complete treatment of security in a single chapter, we have not attempted it. Ultimately, you are responsible for determining the security needs of your system, the suitability of TAO’s feature set to those needs, and the correctness of the implementations in that context. This chapter provides guidance on how to use many of the features available in TAO, but not all.

We have chosen code examples that are faithful to the specification and to TAO’s design objectives. We have chosen not to present examples that demonstrate how to circumvent known issues and problems because such examples would too quickly become irrelevant; for this sort of information we refer you to the release notes included with each source distribution.

29.1.1 Road Map

In this chapter, we explore the topic of security from the perspective of an application with security needs, as well as from the perspective of the features available in TAO’s implementation. While the chapter is designed to be consumed as a whole, you may find it beneficial to read certain sections independently.

If you want to learn more about...

- background information required to make good decisions about using TAO’s security service, Sections 29.2, “Introduction to CORBA Security” and 29.3, “Secure Sockets Layer Protocol” provide overviews on core concepts in CORBA Security and the core technologies in SSL, the security technology used in TAO’s implementation.
• *dealing with certificates* required for using SSL, Section 29.4, “Working with Certificates” discusses the pragmatics of managing certificates for use with SSL and TAO’s security service.

• *building/compiling* TAO’s security service implementation with OpenSSL, look at Section 29.5, “Building ACE and TAO Security Libraries.”

• *code that uses CORBA Security features*, Sections 29.6, “Security Unaware Application” and 29.7, “Security Policy Controlling Application” both give simple examples of applications at various levels of participation in security. Full source code for the examples in this chapter can be found in the TAO source code distribution in \$TAO_ROOT/DevGuideExamples/Security and subdirectories therein.

Please note that in addition to this chapter, in order to fully leverage the power of TAO’s Security Service implementation, you must understand its foundation, the OMG’s Security Service version 1.8 specification (OMG Document formal/02-03-11).

This chapter presents capabilities that allow an application to participate at two levels: *security unaware* and *security policy controlling*. The chapter opens with an introduction to the security service specification. This is followed by an overview of the SSL protocol and a summary of the associated security technology concepts. We then explain how to build the ACE and TAO security libraries, how to configure the ORB’s security services, and how to work with certificates used by SSL. The chapter closes with working examples of TAO’s security features and a summary of the ORB configuration options related to security.

## 29.2 Introduction to CORBA Security

The OMG’s Security Service version 1.8 specification (OMG Document formal/02-03-11) is a comprehensive treatment of security as it relates to distributed object systems and applications. This specification defines a robust feature set and architecture for the implementation of secure, CORBA-based, distributed object systems. The specification:

• Summarizes potential threats faced by a distributed object system.
• Identifies issues arising from the nature of distributed object systems.
29.2 Introduction to CORBA Security

- Describes an abstract Security Reference Model.
- Describes an Implementation Architecture.
- Defines interfaces for use by object system implementers, application developers, and security administrators.

The specification does not, however, describe any specific security technology. Its purpose is to identify the essential features of a secure distributed object system and describe an architecture that can be integrated with current security technologies, such as the Secure Sockets Layer Protocol (SSL), and new security technologies that may emerge in the future.

Allowing application developers to choose the extent to which an application participates in protecting itself is an essential aspect of CORBA Security. There are three levels of participation for an application:

1. **None**: an application may rely completely on the enclosing object system for its protection, remaining ignorant of the existence of any security.
2. **Policy configuration**: an application can set security policies that control the protective measures employed by the object system. However, that application is not obligated to take an active role in the enforcement of that policy, deferring enforcement to the object system itself.
3. **Policy configuration and context inspection**: an application can take an active role by setting policy and by using security context information, obtained at run time, to control access to and use of its services.

This section summarizes essential aspects of the specification:

- The security reference model.
- Feature packages.
- The implementation architecture.

Other aspects of the specification, such as the various application developer interfaces, are discussed in the various examples.

The reference model sets the overall context for CORBA Security. Distributed object systems are deployed under a variety of circumstances and in numerous application domains. Consequently, a distributed object technology, such as CORBA, may be called upon to satisfy a broad range of requirements. The security reference model’s purpose is to define a feature set that satisfies the potential security needs of distributed object systems without specifying a feature solution for any specific system. Some features are purely functional, meant to address the needs of secure applications. Other features are
architectural in nature, meant to give object system implementers flexibility when selecting security technology and implementing security services.

The features described by the reference model are organized into a number of feature packages. Object system implementers balance development costs, system capability, and feature compliance by choosing among the various feature packages.

The implementation architecture describes a means of realizing the reference model in a way that will satisfy the needs of various user communities including:

• End users.
• Application developers.
• System administrators and security managers.
• Object system implementers.

The architecture defines the responsibilities of the various components that must cooperatively realize the features set forth by the reference model and establishes the structural boundaries that separate these components.

29.2.1 CORBA Security Reference Model

The reference model defines a set of features that may be needed by an arbitrary distributed object system. No single set of features is appropriate for all systems. Instead, the feature set summarized here forms a collection of resources from which a system can choose based on its needs.

29.2.1.1 “Principal” Identification and Authentication

Each human user or independently operating software entity, referred to as a principal, that attempts to gain access to or use a secure system shall first establish the right to do so (the CORBA Security concept of principal is completely different from the GIOP 1.0 principal; we only speak of the CORBA Security concept in this chapter). A principal establishes their right to use a secure system by presenting an identity, a claim of “who they are,” and evidence to justify their claim, proof that “they are who they claim to be.” Each principal has at least one identity but may have multiple identities.

For example, a principal may have an identity used to gain access to the system, called an access identity, and a separate identity to represent that same principal in system audit records, called an audit identity. A principal may
have other privileges that further quantify their rights within the system. Identities are referred to as *identity attributes*. Privileges are referred to as *privilege attributes*.

Various mechanisms may be used to identify and authenticate principals. The mechanism used by a particular system may be an integral component but this is not necessarily the case. A secure system may rely on external mechanisms, such as operating system login procedures, to identify and authenticate principals. Most importantly, the model does not define any specific means of identifying and authenticating a principal, and leaves this to be defined by other parties, such as the designer for a specific implementation of CORBA Security, or even an additional OMG specification.

### 29.2.1.2 Delegation of Attributes

Consider the scenario where object $A$, in the course of executing an operation on behalf of client $C$, needs to invoke an operation on another object $Q$. In an unsecure system, this is neither unusual, nor does it pose a problem. However, in a secured system where $C \rightarrow A$ is a secure invocation, there is a question regarding which credentials $A$ should use when performing its invocation on $Q$.

A target object (such as $A$) acting as an intermediary between a principal (such as $C$) that initiates an operation, called an *initiating* or *originating principal*, and another object (such as $Q$) may present its own identity and privilege attributes or it may present those of the initiating principal. In the first case, the intermediary object assumes the role of originating principal with respect to the new target object, somewhat akin to how a deputy temporarily assumes the authority of a sheriff. In the second case, the intermediary object actually mediates between the originating principal and the new target object. Security policies enforced at the new target are based on the identity and privilege attributes presented by the intermediate object.
Simple delegation allows any intermediate object to present the originating principal’s identity and privilege attributes to subsequent targets without restriction. An intermediate object can effectively impersonate the originating principal. Other forms of delegation allow the originating principal to control the extent to which their identity and privilege attributes can be delegated by intermediate objects.

29.2.1.3 Non-Repudiation
A user of a secure system, including a human user, a software component acting on behalf of a human user, and a software component acting independently, shall be prevented from denying their actions within the system. A secure system may generate and record irrefutable evidence of principals’ activities to prevent any principal from denying their actions. This capability requires application level participation and so cannot be provided to security unaware applications.

29.2.1.4 Auditing
Users of a secure system shall be accountable for their actions. Secure systems make and keep records of actions that are relevant to system security. At the system level, these events may include authentication of a principal, a change in security policy, an operation invocation, and so forth. Application level events may also be recorded depending upon an application’s needs. Event records support system security audits by associating each security relevant event with an initiating principal. Audit policies may be needed to restrict the volume of recorded events because the number of security relevant events may be quite large.

29.2.1.5 Transparent Protection
A secure system shall automatically protect all applications, according to applicable security policy, including those applications that choose to remain unaware of security policies and their enforcement. Applications need not take an active role in the specification or enforcement of security policies. Moreover, applications shall not accidentally circumvent security policy through ordinary use of system services. For example, an application should not be able to circumvent security policy, intentionally or unintentionally, merely by invoking an operation via an object reference.
29.2.1.6 **Application Control of Security Policy**

An application may alter security policy. A secure system shall enforce default security policies unless and until those policies are modified by an application. Subsequently, a secure system shall enforce the security policies specified by the application. An application that modifies security policy is not, however, obligated to enforce such policy. The burden of policy enforcement lies with the object system. Of course, modifying a security policy constitutes use of the system. Therefore, any application attempting to modify security policy must have the authority to do so.

29.2.1.7 **Application Enforcement of Security Policy**

Applications may take an active part in the enforcement of security policy. This is particularly true if the object system lacks the mechanisms to enforce a desired policy. A principal’s identity and privilege attributes, represented by a credential, are accessible, at the target, during the course of an invocation. An application may use a principal’s attributes, obtained from the credential, to manage that principal’s use of the system.

29.2.1.8 **Secure Messaging**

Principals invoking operations from remote systems shall be subject to identification and authentication. During a secure invocation, secure object systems first establish a secure association between the client and target. This association includes sufficient context information to enable security policy enforcement, throughout the invocation, by client and server ORBs. Secure systems shall therefore provide a means for identifying and authenticating any principal invoking an operation from a remote system. Moreover, messages exchanged between systems within the context of a secure association may be protected from unauthorized disclosure and tampering.

29.2.1.9 **Administration**

A secure system shall provide a means for administering security policies. Objects are organized into domains for the purpose of defining and administering security policies. Policies are defined with respect to a domain. Policies are applied to objects based upon the domain or domains of which they are a member.
29.2.2 CORBA Security Feature Packages

Note This section refers to the now-obsolete SecurityLevel1 and SecurityLevel2 modules (see “Preface” on page -1065), but the information contained herein remains relevant to the discussion of the general features demanded by the CORBA Security Specification.

As mentioned in 29.2.1, no single set of features is appropriate for all systems, just as there’s not a single set of features available on a particular model car. The specification permits leeway for an implementation of CORBA Security to provide only those features of the reference model (described in 29.2.1.1 through 29.2.1.9) it deems relevant to its target audience. However, for convenience, the specification defines certain “feature packages” with common groupings of reference model features. Feature packaging permits implementations with varying degrees of conformance to the specification.

Table 29-1 CORBA Security Feature Package Quick Reference

<table>
<thead>
<tr>
<th>Package Type</th>
<th>Package Name</th>
<th>In TAO?</th>
<th>What’s this?</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Level 1</td>
<td>✔</td>
<td>Simple, basic security options, such as secure communication. Defined in Security and SecurityLevel1.</td>
<td>29.2.2.1</td>
</tr>
<tr>
<td>Optional</td>
<td>Non-Repudiation</td>
<td>✗</td>
<td>Supports generation of non-repudiation evidence. Defined in NRMODULE.</td>
<td>29.2.2.2</td>
</tr>
<tr>
<td>Security Replaceability</td>
<td>ORB Services</td>
<td>partial</td>
<td>Security implementation provided via interceptors and not in the ORB Core.</td>
<td>29.2.2.3</td>
</tr>
<tr>
<td></td>
<td>Security Services</td>
<td>✗</td>
<td>Security implementation provided external to ORB, but not via interceptors.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 29-1 CORBA Security Feature Package Quick Reference

<table>
<thead>
<tr>
<th>Package Type</th>
<th>Package Name</th>
<th>In TAO?</th>
<th>What’s this?</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Secure Interoperability</td>
<td>CSI 0</td>
<td>✔️</td>
<td>Security policies based on identity only. No privileges attributes transmitted or support for delegation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CSI 1</td>
<td>✔️</td>
<td>Security policies based on identity only. No privilege attributes transmitted. Delegation optional, but unavailable in TAO.</td>
<td>29.2.2.4</td>
</tr>
<tr>
<td></td>
<td>CSI 2</td>
<td>✗️</td>
<td>Security policies based on identity and privileges. Privilege attributes transmitted. Delegation required, and is application-controlled.</td>
<td></td>
</tr>
<tr>
<td>Interoperability</td>
<td>SECIOP</td>
<td>✗️</td>
<td>ORB supports SECIOP.</td>
<td>29.2.2.5</td>
</tr>
<tr>
<td></td>
<td>SECIOP + DCE-CIOP</td>
<td>✗️</td>
<td>ORB supports SECIOP and DCE-CIOP.</td>
<td>29.2.2.6</td>
</tr>
<tr>
<td>Security Mechanism</td>
<td>SPKM/GSSAPI</td>
<td>✗️</td>
<td>Public-key protocol allowing identity-based policies. No delegation support. Requires SECIOP.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GSS Kerberos</td>
<td>✗️</td>
<td>Secret-key protocol allowing identity-based policies. Unrestricted delegation supported. Requires SECIOP.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CSI ECMA</td>
<td>✗️</td>
<td>Secret- and public-key protocol allowing identity- and privilege-based policies. Controlled delegation supported. Requires SECIOP.</td>
<td>29.2.2.7</td>
</tr>
<tr>
<td></td>
<td>SSL</td>
<td>✔️</td>
<td>Public-key protocol allowing identity-based policies. No delegation support. Cannot use SECIOP.</td>
<td></td>
</tr>
</tbody>
</table>
29.2.2.1 Main Functionality Packages

**Level 1**
This package provides the most basic set of security capabilities. Security features are provided to all applications whether or not they participate in security policy enforcement. Principals are identified and authenticated although not necessarily by the ORB.

The ORB provides for secure communication between client and server. Secure communications features include establishing trust between client and server and protecting messages from disclosure and tampering. Appropriate access controls are applied to all secure invocations.

Interfaces and data types are defined in the Security and SecurityLevel1 modules.

Other capabilities and constraints include:

- An intermediate object can delegate the originating principal’s credentials or present its own credentials to subsequent targets.
- Applications participating in security policy enforcement can access principals’ credentials via the operation `SecurityLevel1::Current::get_attributes()` (discussed later in this chapter). Attribute values obtained via this operation may then be used to control access to and use of the system, at the invocation level, and to capture information for audit events.
- Level 1 provides control of security policy via the operation `CORBA::Object::set_policy_overrides()` (also discussed later in this chapter).
- Security administration interfaces are omitted at Level 1.
- Other feature packages, including the Optional and Security Replaceability packages may also be supported at Level 1.

**Level 2**
This package provides additional interfaces for accessing principals’ credentials and additional capabilities for security policy control and administration.

Interfaces and data types are defined in the SecurityAdmin and SecurityLevel2 modules in addition to the Level 1 modules.
29.2.2.2 Optional Packages

Non-repudiation
This is the only optional package. This package provides features and interfaces to support the generation of non-repudiation evidence.

Interfaces and data types are defined in the NRModule.

Note
TAO does not support non-repudiation, so this package is not discussed in this chapter.

29.2.2.3 Security Replaceability Packages

There are many ways to integrate a CORBA Security implementation with an ORB. An obvious way is to tightly couple the two such that the ORB implementation does not stand independent of the Security implementation. However, if more flexibility is desired perhaps because the provider of the ORB implementation is different from the provider of the Security implementation, there must be looser coupling.

The specification provides two standard packages to enable looser coupling at different levels. These packages specify whether or not the ORB’s security services can be replaced and, if so, the mechanism for incorporating additional or alternative services. The specification does not define how the replacement actually occurs, e.g., loading shared libraries, recompilation of source, etc.; these details are left open to the ORB implementer.

The two specified packages are:

- **ORB Services Replaceability**: the ORB invokes security services via specified interceptors and in a specified sequence. The use of other specific interfaces, such as CORBA::Object::get_policy(), is required in the interceptors’ implementation. This places all security service code outside of the ORB core.

- **Security Services Replaceability**: the ORB invokes security services via specified replaceability interfaces. The use of interceptors is optional. Unlike the ORB Services Replaceability Package, this package allows implementation of security services without specific knowledge of how the ORB works. It is similar to ORB Services Replaceability in that it positions all security service code outside of the ORB core.
29.2.2.4 Common Secure Interoperability (CSI) Packages

These packages classify an ORB’s capabilities regarding security policy definition and the delegation of credentials. Classifying capabilities in this manner allows ORBs to make decisions regarding peer capabilities when establishing secure associations and handling secure invocations.

- **CSI 0**: Security policies are based on identity attributes only. Privilege attributes are not transmitted from client to server. Delegation of credentials is not supported.

- **CSI 1**: Security policies are based on identity attributes only. Privilege attributes are not transmitted from client to server. Delegation of credentials, if supported, is unrestricted. A principal cannot control whether or not an intermediary object delegates its credentials nor the extent to which credentials are delegated.

- **CSI 2**: Security policies are based on identity and privilege attributes. Privilege attributes are transmitted from client to server along with identity attributes. Delegation of credentials is supported. A principal can control whether or not an intermediary object delegates its credentials and, when permitted, the extent to which delegation occurs.

29.2.2.5 SECIOP Interoperability Package

This package specifies that the ORB generates and uses security information in IORs and exchanges messages with other ORB’s via the SECIOP protocol using GIOP/IIOP (with specified security enhancements).

29.2.2.6 SECIOP + DCE-CIOP Interoperability

An ORB that realizes this package supports SECIOP over GIOP/IIOP and secure communications, based on DCE security services, via the DCE-CIOP protocol. ORBs that support this package meet all requirements for standard secure interoperability.

29.2.2.7 Security Mechanism Packages

- **Simple Public Key GSS-API Mechanism (SPKM) Protocol**. This protocol allows identity based policies, without delegation, using public-key cryptography. (Public key cryptography techniques are
described later in this chapter.) This protocol requires the SECIOP extensions to the IIOP protocol.

- **Generic Security Service (GSS) Kerberos Protocol.** This protocol allows identity-based policies with unrestricted delegation using secret key cryptography. This protocol also requires the SECIOP extensions to IIOP.

- **CSI ECMA Protocol.** This protocol supports identity- and privilege-based security policies with controlled delegation. Secret and public key cryptography techniques are used in this protocol. This protocol also requires the SECIOP extensions to IIOP.

- **Secure Sockets Layer (SSL) Protocol.** SSL allows identity-based policies without delegation. SSL is independent of GIOP/IIOP and so does not require SECIOP extensions to IIOP.

---

**Note**  *TAO’s security features are based on SSL.*

---

### 29.2.3 CORBA Security Implementation Architecture

The specification addresses the security architecture in four respects:

1. Basic environmental protection and communications.
2. Application components.
3. ORB Core, ORB Services, and Communications Protocols.

#### 29.2.3.1 Basic Environmental Protection and Communications

Basic environmental protection refers to matters such as the host platform, the operating system, the physical plant, and so forth, that are outside the scope of
any software system. As shown in Figure 29-1, a secure system operates within a processing environment.

![Figure 29-1 System Environments](image)

**Figure 29-1 System Environments**

The processing environment consists of the host, the operating system, and other system and application software. A secure system often depends upon the processing environment’s security in one way or another. For example, a secure system may rely on the processing environment to identify and authenticate users by trusting user credentials obtained from or provided by the processing environment.

Every processing environment exists within a physical environment. Different physical environments offer varying degrees of protection. Consider the different levels of protection offered by:

- A street-side kiosk.
- A home office.
- An ordinary office building.
- An office protected by an electronic access control system.
- A secure room protected by armed security guards and multiple electronic access control systems.
A street-side kiosk environment offers little if any protection. An isolated room surrounded by armed guards and multiple access control systems offers a great deal of protection. In each of the examples cited here, the protection provided by the physical environment is an important factor to consider when choosing a processing environment. In turn, the degree of protection afforded by a processing environment is an important factor in the design of a secure software system.

Most distributed systems depend upon interactions between components that are housed by separate processing environments. Many distributed systems involve interactions among multiple components housed by separate processing environments that are located in separate physical environments. Secure communications facilities offer protection from disclosure of and tampering with messages exchanged between systems using those facilities. However, many systems are deployed in circumstances where secure communications facilities are impractical or simply unavailable. Therefore, a secure system may employ cryptographic technology for the express purpose of protecting inter-process communications.

Compromising a secure system may require penetration of the physical and processing environments, penetration of the communications facilities, or both. Furthermore, users who are authorized to enter the physical environment, access the processing environment, and use a system may compromise the system accidentally or perhaps deliberately. Consequently, in addition to protective measures taken by the system itself, action is also required to mitigate threats to the physical and processing environments. The CORBA security architecture is limited to the needs of software systems. It does not address the issues related to processing environments, physical environments, or secure communications facilities.

### 29.2.3.2 Security Architecture

Figure 29-2 depicts the fundamental relationships among:

- Application layer components.
- The ORB Core.
- ORB Services, including security services (note that “ORB Services” are not equivalent to “CORBA Services”, such as CosNaming).
- Communications protocols.
- Security technology components

**Figure 29-2 Security Architecture**

The ORB Core provides the application layer, depicted here by CLIENT, with a coherent representation of a distributed object system. From the application’s perspective, a client object simply invokes an operation on a target object via an object reference. The ORB Core arranges for communications between the client and the target objects as needed to carry out the requested operation. The ORB Core also has the responsibility to invoke ORB Services as each request is processed. This responsibility includes invoking security services as specified by applicable security policy.
When message protection policy requires message integrity or confidentiality, the ORB uses a secure communications protocol to establish trust with its peers, and to protect messages exchanged with its peers. The separation of the ORB core, ORB services, and communications protocols is not evident to the application layer. The different policies that apply to an object reference control the way the ORB processes an invocation, but not the way a client makes an invocation.

When a client invokes an operation on an object reference, the client-side ORB selects the security services applied at the client. The server-side ORB selects the security services applied at the target. If the policies or capabilities at the client and target are inconsistent or incompatible, the server may negotiate a different set of capabilities with the client prior to processing the request or may refuse to process the request altogether.

A security aware application, i.e., an application that controls security policy or participates in the application of security policy, may invoke security services directly as depicted in Figure 29-3.

Security aware applications can alter security policy via CORBA::Object::set_policy_overrides() and obtain references to security services via CORBA::ORB::resolve_initial_references().

Note that the security architecture maintains a clear distinction between:

- Security services and the underlying security technology.
- Secure protocols and the supporting security technology.

This allows ORB implementers to choose a security technology most appropriate to their needs and their target audience. It also affords an opportunity for growth as security technology evolves and new security technologies emerge. The specification does, however, address the integration of specific security technology, such as SSL, with the ORB and how that effects the ORB’s compliance with the specification.

It is also important to be aware that the architecture does not specify interfaces for applications to access the security technology used by the ORB. This does not prohibit an application’s use of such technology. It simply places the responsibility for such interfaces outside the scope of the ORB.
Before a client can invoke an operation on a target object, a security association must be established between the client and the target object through a specific object reference. This association is sometimes referred to as the binding between the client and the target as shown in Figure 29-4. The ORB is responsible for creating and managing security associations. A security association’s lifetime may not exceed that of the process in which it was created although it may be shorter. Security policy and other environmental factors specify how the ORB creates secure associations.
29.2 Introduction to CORBA Security

Context information specific to the client’s use of the object reference is associated with the binding and represented at both the client and target. Some of this context information can be accessed at the target via security services objects such as the Current objects. However, this information is not accessible by the client.

29.2.3.4 TAO’s Security Service Architecture

Note Some figures and text in this section refer to SecurityLevel1 and SecurityLevel2 Current objects. For reasons explained in the Preface, these do not exist in TAO 1.4a. However, we leave them in this sections’ discussion as the concept remains relevant in SecurityLevel3.
TAO’s security service architecture, depicted in Figure 29-5, is consistent with the OMG’s specification.

TAO employs SSL as the underlying security technology. SSLIOP is a secure communications protocol that uses SSL to establish secure associations and to provide message protection.

Figure 29-5 TAO’s Security Service Architecture
Security context information is made available at the target via several objects:

- SecurityLevel1::Current
- SecurityLevel2::Current
- SSLIOP::Current

References to these objects are obtained via
CORBA::ORB::resolve_initial_references().

**Note** SSLIOP::Current is a TAO-specific extension unaffected by the Security Level changes.

Security policies that control the establishment of trust between client and target, and the level of message protection may be set by an application via
CORBA::Object::set_policy_overrides().

## 29.3 Secure Sockets Layer Protocol

SSL supports secure communications between two endpoints, including peer authentication, message integrity, and message confidentiality. **Authentication** guarantees that a peer’s identity is genuine. **Message integrity** guarantees that messages cannot be modified in transit without detection. **Message confidentiality** guarantees that only a message’s intended recipient can read it.

SSL uses both public-key and secret-key cryptography techniques. Public-key techniques protect information exchanged between client and server to establish a secure session. Secret-key cryptography protects messages exchanged after a secure session is established. Public-key cryptography provides better authentication and key exchange mechanisms than secret-key cryptography, but is more computationally intensive and therefore less efficient. Secret-key cryptography is better suited to bulk message protection.

There are three versions of SSL, referred to as SSLv1, SSLv2, and SSLv3. There is also a closely-related specification, Transport Layer Security (TLS), which is based upon SSLv3 and similar in capability. The difference between SSL and TLS is that TLS is a standard from the Internet Engineering Task Force, while SSL is a _de facto_ standard originally defined, published, and
implemented by Netscape Corporation. Throughout this chapter, when we refer to SSL we are referring to SSLv3, but the discussions are applicable to TLS as well. OpenSSL, the SSL implementation used by TAO, supports both SSLv3 and TLS.

This section opens with a conceptual introduction to secret-key and public-key cryptography. It goes on to describe SSL’s architecture and its various protocol elements. The section concludes with an example SSL session. This material is introductory only. It is not a complete treatment of cryptography or SSL. It provides conceptual support for topics covered throughout the remainder of this chapter. If you are already familiar with these topics, you can skip ahead to 29.5.

### 29.3.1 Secret-Key Cryptography

Secret-key cryptography, sometimes called symmetric encryption, is based on a single key that is known by a message’s originator and its receiver. As shown in Figure 29-6, a message’s originator converts the message from a legible form, called plaintext, to an illegible form, called ciphertext, using the secret key and a mathematical algorithm. The message’s receiver uses the same key and a companion algorithm to convert the ciphertext into the original plaintext. A third party attempting to eavesdrop on the exchange between originator and receiver cannot read the message because it is encrypted.

![Figure 29-6 Symmetric Encryption/Decryption](image)

Secret-key cryptography provides message confidentiality only. There is no inherent mechanism to detect that a message was modified in transit. Secret-key technology also carries with it the problem of key exchange. A message originator cannot make a message private unless the message’s receiver knows the key. Therefore, the key must be exchanged in advance,
preferably using a communications medium other than that used to exchange encrypted messages.

Some secure protocols, Kerberos for example, resolve the key exchange problem using a third-party system called a *key server*. A key server knows the secret key of each system that it supports. A message originator first asks the key server for a key and identifies the intended recipient. The key server sends the message originator a *ticket* that contains a new key and a message for the intended recipient. The key server encrypts the enclosed message, which also contains the new key, using the intended recipient's key. The key server encrypts the ticket using the message originator's key. The message originator now has the new key and a private means of conveying it to the intended recipient.

### 29.3.2 Public-Key Cryptography

Public-key cryptography uses a different encryption scheme than secret-key cryptography. Public-key cryptography also includes additional mechanisms that provide for message integrity, identity authentication, and key exchange at run time. The following sections describe both the encryption method and additional capabilities offered by public-key cryptography.

#### 29.3.2.1 Asymmetric Encryption

Public-key cryptography is based on *asymmetric encryption*, which uses two related keys called a *key-pair*. As shown in Figure 29-7, data encrypted using one key can only be deciphered using its companion key and vice versa.

![Figure 29-7 Asymmetric Encryption/Decryption](image-url)
A key-pair’s owner makes one key public and keeps the other key private. A message encrypted using a party’s public key can be decrypted only with that party’s private key. A message originator can send a private message by encrypting it using the receiver’s public key. Only a holder of the receiver’s private key can read such a message. A message originator can also encrypt a message using its private key. This does not prevent disclosure of the message, because any party holding the originator’s public key can read it, but it deters tampering with the message. Only a holder of the originator’s private key can create a message that deciphers correctly using the corresponding public key.

**Message Integrity**

Message integrity is provided through the use of message digests. A message digest is an algorithm that yields a fixed length string from an arbitrary length message. Ideally, a digest would be unique based on the input message, but this is not practical. Therefore, most digest algorithms are devised so the probability of two input messages producing the same digest is very low.

Since more than one input message can yield the same digest, it’s not possible to derive the input message from the digest alone. Although not unique to a particular message, a message digest can effectively serve as a message’s fingerprint when it is enclosed with the message from which it was generated. A message’s receiver can test the message’s integrity by computing a digest and comparing it with the digest that was enclosed with the message. If the two digests are identical, the message was received intact. Otherwise, the message was modified in transit and is not trustworthy.

In practice, message digests are used to compute message authentication codes and to create digital signatures. A message authentication code (MAC) is computed using a message and a key. MACs have the same properties as message digests:

- They cannot be deciphered to discover the original message or the key.
- It is hard to produce two messages that yield the same MAC.
- They can be used to verify a message’s integrity.

A digital signature confirms the authenticity of a message’s originator. A digital signature is created by encrypting a message’s digest (or MAC) using the message originator’s private key. The message’s receiver first obtains the digest using the originator’s public key and then compares it with a locally computed digest.
generated digest. If the two digests are comparable, it proves that the public key’s owner sent the message and that the message was received intact. Moreover, the signature is applicable only to a single message; it cannot be used to forge other messages.

**Identity Binding and Certificates**

Public-key cryptography relies on the association between a key-pair’s public and private components to verify identities. A message that is successfully deciphered using a known public key implies that the originator holds the private key, and therefore that the message was originated by the public key’s owner. For this reason, it is crucial to bind a key-pair’s owner with the owner’s identity.

A *certificate* binds a key-pair holder to his public key. To obtain a certificate, an individual presents substantive proof of their identity to a certificate issuing authority. The *certificate authority* (CA), once satisfied of the requestor’s identity, creates and signs an electronic document containing:

- The certificate holder’s identity.
- The certificate holder’s public key.
- The certificate issuer’s identity.

Identities are represented by a *distinguished name*, a hierarchical naming structure defined by the X.509 specification that gives each entity a unique identifier. The digital signature affixed to a certificate attests to the issuing authority’s identity. A certificate holder offers their certificate as proof of his identity and a means of conveying his public key.

**Certificate Authorities**

A certificate is accepted as genuine if it is signed by a trusted certificate authority. A certificate is authenticated by obtaining the issuing authority’s public key, from the issuing authority’s own certificate, and using it to decipher the signature. If the signature is authentic, the indicated authority issued the certificate. If the system trusts that issuing authority, then the certificate is considered authentic and the owner’s identity accepted. Otherwise, the issuing authority’s certificate is authenticated using the same procedure. This recursive process results in a chain of certificates, all of which are subject to acceptance or rejection.
Certificate authorities vouch for the identity of others. This includes situations where one certificate authority vouches for another. However, as a practical matter, some certificate authorities are not subject to authentication. These authorities vouch for themselves and sign their own certificates. When a self-signed certificate is encountered during the authentication process, the process has reached the decision point. If the system trusts the authority that issued the self-signed certificate, it accepts the entire chain of certificates. Otherwise, it rejects them all.

29.3.3 SSL Architecture

SSL, depicted in Figure 29-8, consists of four separate protocols:

- The Handshake Protocol.
- The Alert Protocol.
- The Record Protocol.

The Handshake Protocol establishes a secure session between two endpoints. During the process, the handshake protocol:

- Negotiates the protocol version.
- Negotiates the cipher suite, the set of cryptographic algorithms used during message exchange.
- Exchanges keys.
- Authenticates peers during the handshake process.

The record protocol processes packets, according to the cipher suite, as they move between upper layer protocols and the transport layer. The alert protocol reports important events such as pending connection closure and processing errors. The change cipher spec protocol signals a change in the current cipher suite.
SSL depends upon a lower level transport protocol to provide reliable message exchange between endpoints. SSL does not have intrinsic support for transport error detection and correction. TCP/IP is the most commonly used transport protocol, although other protocols are sometimes used. TAO has a pluggable protocol called SSLIOP that encapsulates SSL over TCP/IP.

The following sections describe each protocol’s role and present an example SSL session.

### 29.3.3.1 Handshake Protocol

The handshake protocol’s role is to:

- Negotiate a protocol version.
- Negotiate a cipher suite.
- Authenticate peers.
- Establish seed data used to generate secret keys.

This section describes essential aspects of the process. The example session presented later contains a detailed sequence diagram.

Client and server negotiations take the form of a client request followed by the server’s selection. The client indicates the highest protocol version it is capable of supporting and the server replies with the selected version. The client sends the server a list of the cipher suites it is capable of supporting, listing its preference first on the list, and the server responds with the selected
cipher suite. There are no rules that govern the server’s selection mechanism; this is left to the implementers.

A cipher suite defines a combination of server authentication method, key exchange method, digest algorithm, and bulk encryption algorithm. Cipher suites are defined by the protocol specification, they are not arbitrary. A session’s cipher suite informs the record protocol what protective measures to apply to the packet stream between client and server. It is initially empty, which indicates that no protective measures are applied. However, critical information exchanged during negotiations between client and server is protected using public-key cryptography techniques and therefore confidential even when the cipher suite is empty.

To establish the bulk encryption keys the client and server first exchange two chunks of random data. The client generates a chunk and sends it to the server, and vice versa. Next, the client computes the pre-master secret, a new value based on both random data chunks, and sends it to the server. Client and server then compute the various encryption keys using both chunks of random data and the pre-master secret. The resulting keys do not need to be exchanged since client and server have the same seed data and use the same algorithms.

To protect against certain kinds of attack, client and server exchange MACs that are computed over all messages exchanged during the handshake protocol. This guarantees that none of the messages exchanged during the handshake process can be altered without detection.

29.3.3.2 Record Protocol

The record protocol applies protective measures to packets prior to their transmission and reverses the process for received packets. Figure 29-9 depicts the downstream process, the protection of application packets moving toward the transport layer.
The protocol first re-organizes application layer packets into limited-size fragments. Packet boundaries are not preserved. A packet may be split into multiple fragments or combined with other packets to form a single fragment. A MAC is computed and appended to each fragment. The fragments are then encrypted, a header is prefixed, and the resulting record is passed to the transport layer.

Records moving up from the transport layer, depicted in Figure 29-10, are decrypted, authenticated, and passed to the appropriate upper layer protocol.

Figure 29-9 Downstream Packet Processing
The record protocol does not differentiate between upper layer protocols with respect to record protection. Protective measures, when in force, are applied to all packets including those associated with one of the other SSL protocols.

The SSL specification allows records to be compressed prior to the encryption step but does not specify any compression algorithms. Compression is not discussed in this chapter because it is not supported in many SSL implementations, although it is supported by OpenSSL.

29.3.3.3 Change Cipher Spec Protocol

The change cipher spec protocol consists of a single message. It is sent during the handshake protocol after the session parameters have been negotiated. It indicates that the sender will apply the protective measures specified by the cipher suite to all subsequent messages.
29.3.3.4 Alert Protocol

The alert protocol serves two purposes. A close_notify warning signals its receiver that the sender is closing the connection. All other messages indicate that the sender encountered an error condition.

29.3.3.5 Example Session

Figure 29-11 depicts a typical SSL session:

1. **ClientHello**: the client initiates the handshake protocol by sending the ClientHello message. This message indicates the highest protocol version supported by the client, lists the cipher suites supported by the client with the client’s preference appearing first, and contains a chunk of random data.

2. **ServerHello**: the server responds to a ClientHello with a ServerHello. This message contains the server’s selections for protocol version and cipher suite. It also contains a chunk of random data.

3. **Certificate (to client)**: the Certificate message contains a list of ASN1-encoded certificates. The server’s certificate appears first. Each subsequent certificate identifies the authority that issued the preceding certificate. The server’s certificate is always sent to the client unless the chosen cipher suite does not require it.

4. **ServerHelloDone**: this message indicates that the server has completed this phase of the handshake protocol.

5. **ClientKeyExchange**: this message contains the pre-master secret or, for some cipher suites, data needed to compute the pre-master secret.

6. **ChangeCipherSpec (to server)**: this message indicates that all subsequent messages will be protected according to the chosen cipher suite.

7. **Finished (to server)**: this message contains a MAC that was computed over all preceding handshake messages.

8. **ChangeCipherSpec (to client)**: this message indicates that all subsequent messages will be protected according to the chosen cipher suite.

9. **Finished (to client)**: this message contains a MAC that was computed over all preceding handshake messages.

A secure session is now established between client and server.

10. The client and server exchange application data.

11. **Alert**: close_notify: this message informs the server that the client is closing the connection.
The connection is closed.

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientHello</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ServerHello</td>
</tr>
<tr>
<td></td>
<td>Certificate</td>
</tr>
<tr>
<td></td>
<td>ServerHelloDone</td>
</tr>
<tr>
<td>ClientKeyExchange</td>
<td></td>
</tr>
<tr>
<td>ChangeCipherSpec</td>
<td></td>
</tr>
<tr>
<td>Finished</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChangeCipherSpec</td>
</tr>
<tr>
<td></td>
<td>Finished</td>
</tr>
<tr>
<td>=== Session Established</td>
<td></td>
</tr>
<tr>
<td>{application data}</td>
<td></td>
</tr>
<tr>
<td>Alert: close_notify</td>
<td></td>
</tr>
<tr>
<td>===Connection Closed</td>
<td></td>
</tr>
</tbody>
</table>

Figure 29-11 Example SSL Session
29.4 Working with Certificates

To develop and test applications that use TAO’s security services, you will need to create certificates. OpenSSL includes a tool called openssl for creating, manipulating and inspecting keys, certificate requests, and certificates. OpenSSL also includes a Perl script called CA.pl to facilitate certain operations, including in particular those operations supporting the creation and management of certificates. Of interest here are the methods for:

• Creating your own certificate authority.
• Generating certificate requests.
• Creating and signing certificates.

Certificates may appear in either ASN1 or Privacy Enhanced Mail (PEM) format. CA.pl’s default behavior produces certificates in PEM format and all certificates presented in this chapter appear in this format.

Note Some OpenSSL packages vary the name and/or location of this script slightly. You will have to determine the location of the script in your installation of OpenSSL.

In the examples that follow:

user input is presented in this font
system output is presented in this font

29.4.1 Environment Setup

openssl, and therefore CA.pl, use a configuration file to specify certain options. The path to a suitable configuration file may be passed as an option to many openssl commands. The configuration file can also be specified by an environment variable called OPENSSL_CONF.

The source distribution includes a default version of the aforementioned configuration file called openssl.cnf. The location of this file varies with the installation. On UNIX and UNIX-like platforms, you may find it in the ssl directory that appears in the path to CA.pl. On Windows, you will find it in the release’s apps directory.
The examples assume use of the default configuration file and the
OPENSSL_CONF environment variable.

29.4.2 Create Certificate Authority
To create a new local certificate authority use:

$ CA.pl -newca

Here is an example:

$ CA.pl -newca
CA certificate filename (or enter to create)CR
Making CA certificate ...
Using configuration from /usr/local/ssl/openssl.cnf
Generating a 1024 bit RSA private key
..............................+++++
....................+++++
writing new private key to './demoCA/private/cakey.pem'
Enter PEM pass phrase:taosecurity
Verifying password - Enter PEM pass phrase:taosecurity
-----
You are about to be asked to enter information that will be incorporated into your
certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Missouri
Locality Name (eg, city) []:St. Louis
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Object Computing, Inc.
Organizational Unit Name (eg, section) []:TAO
Common Name (eg, YOUR name) []:John Smith
Email Address []:smith_j@ociweb.com
$  

This command:
- Builds a local directory hierarchy, ./demoCA, for use by the other
certificate authority commands.
- Creates the CA’s private key.
- Stores the CA’s private key in ./demoCA/private/cakey.pem.
- Creates the CA’s self-signed certificate.
29.4 Working with Certificates

- Stores the CA’s certificate in ./demoCA/cacert.pem.

During this process, you will be prompted for the CA’s identity attributes and a pass phrase to guard use of the CA’s certificate. You will be prompted for the CA’s pass phrase each time you sign certificate requests.

You are now ready to create a certificate request.

29.4.3 Create Certificate Request

To create a new certificate request, use:

```
$ CA.pl -newreq
```

Here is an example:

```
$ CA.pl -newreq
Using configuration from /usr/local/ssl/openssl.cnf
Generating a 1024 bit RSA private key
....+++++
.........+++++
writing new private key to 'newreq.pem'
Enter PEM pass phrase:taosecurity
Verifying password - Enter PEM pass phrase:taosecurity
-----
You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.'; the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Missouri
Locality Name (eg, city) []:St. Louis
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Object Computing, Inc.
Organizational Unit Name (eg, section) []:TAO
Common Name (eg, YOUR name) []:John Smith
Email Address []:smith_j@ociweb.com

Please enter the following 'extra' attributes to be sent with your certificate request
A challenge password []:corbasecurity
An optional company name []:OCI
Request (and private key) is in newreq.pem
```

$
This command:

- Generates a new private key.
- Generates a new certificate request.
- Stores the certificate request and private key in `newreq.pem`

During this process, you will be prompted for the user’s attributes and a pass phrase to protect the user’s certificate. The completed request document will appear similar to the following:

```plaintext
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,CB2F8D7A7D2FFA06

FTVP97A35qWIuITZctfEqP2J1ldwsKttmlj16c1PTyfWmTk3439CuXI7OFgX09ME
A1dG2eEnFCWI11M0Xvpy/GKn0q5pfeNYvCiO/JjkRtQSwJO+xgLdSk7pluUA6b
uiUValCevBslYe19g9UWH9KhUvBrtrICedIK5e+d97ed3A5GEZXF8f8+jU/FM5M
sA4a+FT2uq2SHj47rXTfBpcevmYZjoTEUzfrovedqCFLhf9FfMyODRe9y2Fq5g
yk87GY5mIpCooDrqKspbg9Is3yFXXn1d8jyQxgHpJzyQVedi4Gz3Y1iPHM4K2G
Xy91A6L+0NZuZRMALNUjQweiJ1fFrkRJ1Bc5nmy010aieqRLy17ju7rTQVA5xF/
YQQcUrMugHAm0u9n4+uLVdehgnlmp0la+EjHr7ETxnUQyQrnrDvOuCMynFDmuYLo
2fSu90iszuxZx1F1PtzfeCob/IA5Jpf/NA3HxRxDn390+MXnxAVdm2zO/JxPVzmeCdZ
Mtmjhg+fujJ0c9X9eIvWj73v9xBLFv7FhMZXUnrNHYikCvBv39WcE8eJ12I
EI1cDluClVscD4PD5WdMYdMhFh6yAFL0xRBE8e8vCn2eEXTU0QXzHARG2fW9f1n
4skNvWcsG5FpFJe3xBEQ02QwQxwO8tntF1BWOzDnqKvHC6v5HmotIwevjYX9ZCe8vWn
UNcNjwUFRZ8s75ZrVzBB6NAF4q+UUB6N+EQBCsfHYtleyeuDwhuafjxKz11x4c
rXldK9t2ufCwpnLYLz1g/MzaTyCay6ddXntUs2DueHr0FCmAy8I9g==
-----END RSA PRIVATE KEY-----

-----BEGIN CERTIFICATE REQUEST-----
MIICDjCCAxCCAQAw2szxXaJbNhvNBYATA1VMREWdwYDVQQIEdwaNaXNz3YvaTES
MBAGA1UEBxMvuM3JXvMDR8wQ9DYVQQIExZ5Ymp1b3Qg29tcmVoA657nLCBJ
bMVnMnwQgYDVQQLExNQUU8x7zEzARBgNBAMTCkpvag4gU1pdgXU7IAF8BQgkhiL
9w0QCQEQWxhXreaR0x2pAb2NpD2V1LMvBCTBnxANBoqkgqkhg/kgw0BAEQEAAbQjAw
gYkcqYEaqzIMMcYMkHifyQP8GB9PNbARx04Yt0Mdm6q7shbU/i1qndGo8ygwUdcG1
cxF0dm6PLLJzuz5XysuiIGfVBsCD68b2QxP9hyCA3jxmg9MYwAw61KxhFX4wWWY
EPwBxazzJ9117ioo7yVHovc02bpCuphho/XIL1Odxd1HG8g1PC3sCwEAIAyMBIG
CQsGaISb3QjEJAFENfNP0kwhHAYJKOZIhvcNAQkHMQ87DUNVcmJh2cVjdpPhHkw
DQYJK0ZIhvcNAQEBQADgYEAcpg+VrzdThVUE+buFwAzc9LwHgS5xade9wU51Wp
SwbixoC6MP+UKx6BqvjYNTWCJEXGm7Ikhr34w6SWkxozTM+bn1AS+0vX2ezXmeU
jXtvTNTr2je5Xv9k3e/CyQPCICVGZTJThRJuFB4I2Do2FtydLT8AOX9+Acz
daA=
-----END CERTIFICATE REQUEST-----
```

The file must remain as is to serve as a certificate request. The file can also serve as a private key in this form. However, the private key portion can be copied into a separate file. (Lines delimiting the private key are required.)
You are now ready to sign the request and create a certificate.

**29.4.4 Sign Certificate Request and Issue Certificate**

To sign a certificate request and issue a certificate, use:

```
$ CA.pl -sign
```

Here is an example:

```
$ CA.pl -sign
Using configuration from /usr/local/ssl/openssl.cnf
Enter PEM pass phrase:taosecurity
Check that the request matches the signature
Signature ok
Certificate Details:
  Serial Number: 1 (0x1)
  Validity
    Not Before: Sep 9 23:08:46 2005 GMT
    Not After : Sep 9 23:08:46 2006 GMT
  Subject:
    countryName = US
    stateOrProvinceName = Missouri
    localityName = St. Louis
    organizationName = Object Computing, Inc.
    organizationalUnitName = TAO
    commonName = John Smith
    emailAddress = smith_j@ociweb.com
X509v3 extensions:
  X509v3 Basic Constraints:
    CA:FALSE
  Netscape Comment:
    OpenSSL Generated Certificate
X509v3 Subject Key Identifier:
X509v3 Authority Key Identifier:
  DirName:/C=US/ST=Missouri/L=St. Louis/O=Object Computing, Inc./OU=TAO/CN=John Smith/emailAddress=smith_j@ociweb.com

Certificate is to be certified until Sep 9 23:08:46 2006 GMT (365 days)
Sign the certificate? [y/n]: y

1 out of 1 certificate requests certified, commit? [y/n] y
Write out database with 1 new entries
Data Base Updated
Signed certificate is in newcert.pem

This command:

• Processes the certificate request held in newreq.pem.
• Generates a new public key and certificate.
• Stores the certificate in the file ./newcert.pem.

The completed certificate will appear similar to the following:

Certificate:
Data:
  Version: 3 (0x2)
  Serial Number: 1 (0x1)
  Signature Algorithm: md5WithRSAEncryption
  Issuer: C=US, ST=Missouri, L=St. Louis, O=Object Computing, Inc.,
          OU=TAO, CN=John Smith/emailAddress=smith_j@ociweb.com
  Validity
    Not Before: Sep 9 23:08:46 2005 GMT
    Not After : Sep 9 23:08:46 2006 GMT
  Subject: C=US, ST=Missouri, L=St. Louis, O=Object Computing, Inc.,
          OU=TAO, CN=John Smith/emailAddress=smith_j@ociweb.com
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    RSA Public Key: (1024 bit)
      Modulus (1024 bit):
        00:ab:12:0c:00:29:98:28:78:a9:0f:c1:81:f4:f3:
        60:33:d6:0c:c0:0c:3a:d4:ad:el:c4:5e:21:c3:05:
        98:10:fc:01:c7:3c:c9:f7:5d:7b:8a:8a:3b:c9:51:
        17:76:21:07:f2:08:8f:0b:7b
      Exponent: 65537 (0x10001)
  X509v3 extensions:
    X509v3 Basic Constraints:
      CA:FALSE
    Netscape Comment:
      OpenSSL Generated Certificate
  X509v3 Subject Key Identifier:
  X509v3 Authority Key Identifier:
    DirName:/C=US/ST=Missouri/L=St. Louis/O=Object Computing, Inc./OU=TAO/CN=John Smith/emailAddress=smith_j@ociweb.com
Signature Algorithm: md5WithRSAEncryption

```

-----BEGIN CERTIFICATE-----
MIID3zCCA0igAwIBAgIBATANBgkqhkiG9w0BAQQFADCBmAzELMAkGA1UEbXMMcHVw
ETAPBgNVBASBEjJzMTswNDAwMDoGA1UEAwwXU29tZXJ0aW9ucyBhcmdldCBw
L29taXZlLmNvbTAfBgkqhkiG9w0CBQwwHwYDVR0PAQH/BAIiGBkqkgAwHMQ
DQYJKoZIhvcNAQEFBFAnB2ZtMjQ26xOwIBAgYIKwYBBQUHAQCCAQGAMBQAw
GzAJBgNVBAMTB1NlcnRpZmljYXRlMTqAIBhGA1UdEwEB/wQFMA0GA1UdDgQIB
AoBgRMA0GAYAATA4ANBgqQMA0GAYEAMCwGCSqGSIb3DQEBCwUAA4GCSqGSIb3
DQEBAQUAA4GAYEAMCwGCSqGSIb3DQEBAQUAQAQAMzQ6aG9yZW0gVXNlcm5lciBk
YXlpbiBNSU9TcnVzdCwJdQYJKoZIhvcNAQEFBFAnB2ZtMjQ26xOwIBAgYIKwYBB
QwIBAgIHNlcnRpZmljYXRlMB8GA1UdDgwEB/wQFAwIBAgIKwYBBQwIBAByBASg
GCSqGSIb3DQEBCwUAMzQ6aG9yZW0gVXNlcm5lciBkYXlpbiBNSU9JcnVzdCwJd
QYJKoZIhvcNAQEFBFAnB2ZtMjQ26xOwIBAgYIKwYBBQwIBAByBASgGCSqGSIb3
DQEBCwUAMzQ6aG9yZW0gVXNlcm5lciBkYXlpbiBNSU9JcnVzdCwJdQYJKoZIhvc
NAQEFBFAnB2ZtMjQ26xOwIBAgYIKwYBBQwIBAByBASgGCSqGSIb3DQEBCwUAMzQ
6aG9yZW0gVXNlcm5lciBkYXlpbiBNSU9JcnVzdCwJdQYJKoZIhvcNAQEFBFAnB2Z
-----END CERTIFICATE-----
```

This file can serve as a certificate in this form or the certificate portion can be
copied to a separate file. (Lines delimiting the certificate are required.)

### 29.4.5 Removing Key Pass Phrases

By default, keys are protected by the PEM pass phrase provided when the
certificate was created. This pass phrase must be supplied whenever an
application utilizing the key is launched. To run processes in the background,
remove the pass phrase as follows:

```
$ openssl rsa -in key-with-password.pem -out key-without-password.pem
```

---

This file can serve as a certificate in this form or the certificate portion can be
copied to a separate file. (Lines delimiting the certificate are required.)

### 29.4.5 Removing Key Pass Phrases

By default, keys are protected by the PEM pass phrase provided when the
certificate was created. This pass phrase must be supplied whenever an
application utilizing the key is launched. To run processes in the background,
remove the pass phrase as follows:

```bash
$ openssl rsa -in key-with-password.pem -out key-without-password.pem
```
The file `key-with-password.pem` holds a private key protected by a password. The key with password removed is stored in the file `key-without-password.pem`.

### 29.4.6 Certificate Commands Summary

**Environment variables:**
- `PATH` includes directories containing `openssl` and `CA.pl`.
- `OPENSSL_CONF` specifies path to configuration file.
- `openssl.conf` is the default configuration file provided with the OpenSSL source release.

**Commands:**
- `CA.pl -newca` establishes a new local certificate authority.
- `CA.pl -newreq` creates a new private key and certificate request.
- `CA.pl -sign` creates a new public key and a signed certificate.
- `openssl rsa -in key-with-password.pem -out key-without-password.pem` removes the pass phrase from a private key.

### 29.5 Building ACE and TAO Security Libraries

This section discusses construction of executables that use TAO’s security features and the supporting libraries; `libACE_SSL`, `libTAO_Security`, and `libTAO_SSLIOP`.

The ACE SSL library (`libACE_SSL`) wraps OpenSSL’s implementation of SSL in a manner consistent with ACE’s Inter-Process Communication Service Access Point (IPC_SAP) architecture and, consequently, TAO’s pluggable protocol architecture. SSLIOP (`libTAO_SSLIOP`) is a secure communications protocol, founded on the ACE SSL wrappers, which supports:
- Peer authentication.
- Message integrity.
- Message confidentiality.
TAO’s Security services library (libTAO_Security) provides security aware applications with:

- Access to security attributes.
- A means to control message protection level.
- A means to control the establishment of trust between clients and servers.

OpenSSL is the foundation of TAO’s security service. It may already be installed on your platform, particularly if you are working with a Linux platform. However, on many other UNIX and UNIX-like platforms, and on Windows platforms, you may need to obtain, build, and install OpenSSL yourself. OpenSSL source is freely available from the OpenSSL web site: <http://www.openssl.org>. The source release includes some general instructions, the README file, and files with specific instructions for several platforms, including the INSTALL file for UNIX-like platforms and the INSTALL.W32 file for Windows platforms.

The following sections address:

- Building the security libraries.
- Testing the libraries.
- Building executables.

**Note**  
The instructions for building the security libraries have changed since TAO 1.3a.

### 29.5.1 Building Security Libraries: UNIX Variants

ACE and TAO security libraries can be built individually or along with other ACE and TAO libraries. The following settings support construction of the security-related libraries and application executables:

- `SSL_ROOT` (in `platform_macros.GNU` or as an environment variable): specifies the path to the OpenSSL installation directory.
- `ssl` (in `platform_macros.GNU`): indicates whether or not to build the security-related libraries. Specifying `ssl=1` causes the security-related libraries to be built.
• ssl (in MPC’s default.features): must be set to 1 (ssl=1) for MPC to generate project files to build SSL-related projects. See 4.3.2.3 for more information on MPC’s feature file.

For more information on these and other build settings, see Appendix A.

With these settings assigned correctly, you can build the security libraries simply by building the normal TAO libraries. You can also build the TAO libraries individually with the following commands:

```bash
cd $TAO_ROOT/orbsvcs/orbsvcs
make -f GNUmakefile.Security
make -f GNUmakefile.SSLIOP
```

libTAO_SSLIOP depends upon libTAO_Security, so be sure it is built prior to building libTAO_SSLIOP.

### 29.5.2 Building Security Libraries: Microsoft Visual Studio for Windows

TAO’s source release contains pre-defined project and workspace/solution files for building ACE and TAO libraries, including those related to TAO’s security service. In the %TAO_ROOT% directory, TAOACE.sln (Visual C++ 7.1) or TAOACE.dsw (Visual C++ 6.0) includes projects needed to build the various security-related libraries.

The following settings support construction of the security-related libraries and application executables:

• SSL_ROOT (environment variable): specifies the path to the OpenSSL installation directory.

• ssl (in MPC’s default.features): must be set to 1 (ssl=1) for MPC to generate project files to build SSL-related projects. See 4.3.2.3 for more information on MPC’s feature file.

For more information on these and other build settings, see Appendix A.

The MPC-generated project settings establish the appropriate OpenSSL include and library search paths based on the SSL_ROOT environment variable (e.g., $(SSL_ROOT)/include, $(SSL_ROOT)/inc32, and $(SSL_ROOT)/lib), so you do not need to add these locations to the Visual Studio tools options settings.
To build the security libraries, open the `TAOACE.sln` solution file in Visual Studio (open the `TAOACE.dsw` workspace file if you are using Visual C++ 6.0) and simply build the `SSLIOP` project. Since the `SSLIOP` project depends upon the `SSL` and `Security` projects, building this one project will build all the necessary security libraries: `ACE_SSL`, `TAO_Security`, and `TAO_SSLIOP`.

### 29.5.3 Testing the Security Libraries

Tests for TAO’s security services are located in `$TAO_ROOT/orbsvcs/tests/Security`. README files describe each test and provide other useful information such as instructions for running the test manually. A perl script, `run_test.pl`, is provided to execute each test. Makefiles and project files are provided in most, if not all, cases.

### 29.5.4 Building Security Unaware Executables

TAO’s support for security-unaware applications, i.e., the secure communications protocol, is implemented in the `TAO_SSLIOP` library. Thus, a security-unaware application must link with this library if the executable is statically linked. Otherwise, no additional steps are required because the library is loaded dynamically by the service configurator.

MPC projects for security-unaware applications can simply inherit from the `ssl` base project. For example, here is the file for the `SecurityUnawareApp` example in `$TAO_ROOT/DevGuideExamples/Security/SecurityUnawareApp`:

```plaintext
project(*Server): taoexe, portableserver, security, ssl {
  requires += exceptions
  Source_Files {
    Messenger_i.cpp
    MessengerServer.cpp
  }
}

project(*Client): taoexe, security, ssl {
  requires += exceptions
  Source_Files {
    MessengerC.cpp
    MessengerClient.cpp
  }
}
```
**29.5.5 Building Security Aware Executables**

TAO’s support for security aware applications that wish to set security policy or access security context data at run time is implemented in the TAO_Security library and the TAO_SSLIOP library. OpenSSL libraries are also required for security-aware applications that wish to access the security attributes:

- For UNIX and UNIX-like platforms, the OpenSSL libraries are `libssl` and `libcrypto`.
- For Windows platforms, the OpenSSL library is `libeay32`.

Library path environment variables must include paths to all but TAO_SSLIOP. A path to TAO_SSLIOP is only required at compile time for statically linked executables.

MPC projects for security aware applications can simply inherit from the security and ssliop base projects. For example, here is the project file for the ParticipatingApp example in $TAO_ROOT/DevGuideExamples/Security/ParticipatingApp:

```plaintext
project(*Server): orbsvcexe, portableservlet, security, ssliop {
    requires += exceptions
    Source_Files {
        Messenger_i.cpp
        MessengerServer.cpp
    }
}

project(ParticipatingApp_Client): orbsvcexe, security, ssliop {
    requires += exceptions
    exename   = MessengerClient

    Source_Files {
        MessengerC.cpp
        MessengerClient.cpp
    }
}
```

**29.6 Security Unaware Application**

Providing security for applications that do not take an active part in setting or enforcing security policy is largely a matter of configuring the environment,
and configuring and installing TAO’s SSLIOP protocol. The sections that follow explain how to do this:

- Describe how to set up the environment.
- Describe how to configure and load the SSLIOP protocol.
- Present relevant code examples.

### 29.6.1 Environment Setup

There are four separate environment variables that may be used by the SSL libraries depending upon circumstances at runtime:

- **SSL_CERT_FILE**: gives the pathname of a file containing one or more certificates from certifying authorities.

  When this variable is set, the named file is the source of all certifying authority certificates. Refer to 29.6.1.1 for additional information.

  A default value for `SSL_CERT_FILE` is defined by the macro `ACE_DEFAULT_SSL_CERT_FILE` in `$ACE_ROOT/ace/SSL/sslconf.h`.

- **SSL_CERT_DIR**: gives the pathname of a directory housing certificates from one or more certifying authorities.

  When this variable is set, the named directory is the source of all certifying authority certificates. Refer to 29.6.1.1 for additional information.

  A default value for `SSL_CERT_DIR` is defined by the macro `ACE_DEFAULT_SSL_CERT_DIR` in `$ACE_ROOT/ace/SSL/sslconf.h`.

- **SSL_EGD_FILE**: gives the pathname of a random data source generated by the entropy-gathering daemon (EGD).

  OpenSSL needs a source of random data. EGD is a perl script that produces random data and delivers it via a UNIX domain socket. EGD is widely used as a random data source, although it was not tested during the writing of this chapter.

- **SSL_RAND_FILE**: gives the pathname of a data file holding state data from SSL’s pseudo random number generator.

  OpenSSL has its own pseudo-random number generator. When this mechanism is used as the random data source, state information is stored
in the named file between invocations. OpenSSL’s pseudo-random number source was not tested during the writing of this chapter.

All of the examples presented in this chapter employ only a single certifying authority and therefore use the `SSL_CERT_FILE` environment variable to refer to the certifying authority’s certificate. No values were assigned to any of the other environment variables.

### 29.6.1.1 Using Multiple Certificate Authorities

Authenticating a certificate when multiple certifying authorities are involved requires access to several certificates. For example, suppose a principal named Shannon obtains a certificate from a certifying authority named ACME Certificate Co. In turn, ACME obtained its certificate from Certificates Unlimited, Inc., which signs its own certificate and therefore acts as the root certificate authority (refer to Figure 29-12).

To verify Shannon’s certificate, the server’s authentication routines need to examine each certifying authority’s certificate. The certificates may be concatenated into a single file, referred to by `SSL_CERT_FILE`, or they may be housed in a common directory, referred to by `SSL_CERT_DIR`. When a principal’s certificate is authenticated and `SSL_CERT_FILE` is set, the named file is searched for certifying authority certificates. Otherwise, if `SSL_CERT_DIR` is set, the named directory is searched.

To facilitate certificate directory searches, each file is named based upon a hash code generated from the certificate holder’s subject name. For example, the subject name:

```
C=US, ST=Missouri, L=St. Louis, O=Object Computing, Inc., OU=TAO, CN=Certifying Authority/Email=ca@ociweb.com
```

yields a hash value of 53052543. The string “.0” is appended yielding a file name of 53052543.0. A file by that name, which may contain the certificate or may be a link to the certificate file, is placed in the certificate directory.

When authenticating a principal’s certificate:

- The issuing authority’s name is extracted from the certificate.
- The corresponding hash value is computed.
- The certificate is obtained from the certificate directory and authenticated.
The process continues until a trusted certificate is encountered or a required certificate cannot be found.

Figure 29-12 Certificate Authentication Chain
### 29.6.2 Configuring and Loading SSLIOp

The following options to the SSLIOp protocol factory can be used to configure the SSLIOp protocol:

- `-verbose | -v` turns on debugging messages during loading of the SSLIOp protocol library. You will get messages equivalent to passing `-ORBDebugLevel 1` on the command line.

- `-SSLNoProtection` influences how a client submits requests to servers and how a server receives requests from clients. It is equivalent to specifying `DEFAULT:eNULL` for the SSL cipher list.

When SSLIOp gets loaded, default client behavior is to submit all requests via SSLIOp. The default server behavior is to receive all requests via SSLIOp only.

When `-SSLNoProtection` is specified, default client behavior is to submit all requests via IIOP. Default server behavior is to receive requests via either protocol. A process acting as a server now receives requests via IIOP and SSLIOp.

- `-SSLCertificate FORMAT: filename`: gives the pathname for the principal’s certificate file. FORMAT specifies the certificate file’s format; ASN1 and PEM (Privacy Enhanced Mail) formats are supported. (All examples presented in this chapter use certificates in PEM format.)

- `-SSLDhparams FORMAT: filename`: gives the pathname for Diffie-Hellman cipher parameters. While you may specify ASN1 or PEM for FORMAT, SSLIOp only recognizes PEM (ASN1 results in an error).

A Diffie-Hellman cipher (anonymous or otherwise) requires these parameters. Typically, a PEM-encoded DSA certificate includes a DH parameters section. However, because the Anonymous Diffie-Hellman (ADH) cipher uses no certificates, you must use this option (typically on the server side) for ADH.

If you use the `-SSLDhparams` option together with `-SSLCertificate`, SSLIOp uses the parameters specified by this option.

- `-SSLPrivateKey FORMAT: filename`: gives the pathname for the principal’s private key file.

- `-SSLAcceptTimeout timeout-value` gives the amount of time (specified as a floating-point value) that SSLIOp will wait to complete the SSL
handshake (see 29.3.3.1, “Handshake Protocol” on page 1093). Note that
the SSL handshake includes the TCP handshake to establish a TCP
connection as well as the SSL protocol handshake, so you should specify
your value accordingly.

- -SSLAuthenticate which: specifies the level of peer authentication
  conducted as secure sessions are established between client and server.
  Possible values for which are:
    - NONE: disables peer authentication. When this option is set, a process
does not authenticate its peers regardless of its role.
    - SERVER: causes a process acting as a client to authenticate its peers.
    - CLIENT: causes a process acting as a server to authenticate its peers.
      (There are limitations in the SSL protocol that make this setting
equivalent to SERVER_AND_CLIENT.)
    - SERVER_AND_CLIENT: causes a process to authenticate its peers
      regardless of its role, client or server.

Values for these options are defined via a service configuration directive and
passed to the SSLIOP protocol factory by the service configurator. The
SSLIOP factory must be loaded explicitly via a separate directive because, by
default, SSLIOP is not loaded. The SSLIOP protocol factory can be loaded
with the default resource factory or with the advanced resource factory.

Note In TAO 1.3a, using SSLIOP required the use of the -ORBConnectStrategy
blocked client strategy factory option. This option is no longer necessary in
TAO 1.4a. The TAO SSLIOP connector automatically enforces the blocked
connect strategy. Specifying the option will not cause an error. See 22.3.1 for
more information on the -ORBConnectStrategy client strategy factory
option.

29.6.3 Security Unaware Application Example
To demonstrate an application that is protected via TAO’s security service
without the application’s direct involvement, we configure the Messenger
server and Messenger client from 3.3 so that they:

- Authenticate their peers.
- Provide message confidentiality.
• Provide message integrity.

The full source code and configuration files for this example can be found in $TAO_ROOT/DevGuideExamples/Security/SecurityUnawareApp.

The examples that follow require:

• A certifying authority’s certificate named cacert.pem.

• A certificate, signed by the certifying authority, and its corresponding private key, respectively named servercert.pem and serverkey.pem.

• A client certificate, signed by the certifying authority, and its corresponding private key respectively named clientcert.pem and clientkey.pem.

• The environment variable SSL_CERT_FILE containing the path to the certifying authority’s certificate.

Configuring the server is easily accomplished with the service configuration directives found in Figure 29-13. These directives are placed in a service configuration file, here called server.conf, which is used to configure the Messenger server upon invocation as follows:

```
$ MessengerServer -ORBSvcConf server.conf
Enter PEM pass phrase:
IOR written to file Messenger.ior
```

In this example the pass phrase has not been stripped from the private key. Consequently, the user must type the pass phrase each time the server is started.

Invoking the Messenger client without configuring it to use SSLIOP causes an exception:

```
$ MessengerClient
Uncaught CORBA exception: NO_PERMISSION (IDL:omg.org/CORBA/NO_PERMISSION:1.0)
```

Figure 29-13 Security Unaware Server-side svc.conf
The exception indicates that the client does not have permission to invoke the requested operation. Configuring the client is also easily accomplished with the service configuration directives found in Figure 29-14.

```c
Dynamic SSLIOP_Factory Service_Object * \n  TAO_SSLIOP: make_TAO_SSLIOP_Protocol_Factory() \n  "-SSLAuthenticate SERVER_AND_CLIENT \n  -SSLPrivateKey PEM:clientkey.pem \n  -SSLCertificate PEM:clientcert.pem" 
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

**Figure 29-14 Security Unaware Client-side svc.conf**

These directives are placed in a separate service configuration file, which we call `client.conf`, used to configure the Messenger client upon invocation:

```bash
$ MessengerClient -ORBSvcConf client.conf
Enter PEM pass phrase:
message was sent
$
```

The client now reports that the message was sent and the server prints the message:

```
Message from: Chief of Security
Subject:      New Directive
Message:      Implementing security policy now!
```

Failing to name the certifying authority’s certificate via the `SSL_CERT_FILE` environment variable is a common mistake, and results in a different exception:

```bash
$ MessengerClient -ORBSvcConf client.conf
Enter PEM pass phrase:
ACE_SSL (784|1040) error code: 336134278 - error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
Uncaught CORBA exception: TRANSIENT (IDL:omg.org/ORB/CORBA/TRANSIENT:1.0)
$
```

This exception is less helpful, indicating no more than a problem with a transient object reference. The error message from the ACE SSL wrappers is more helpful in that it indicates an error related to certificate authentication. In the case at hand, authentication of the server’s certificate failed because the certifying authority’s certificate was unknown to the client.
However, if the client application is concerned only with message confidentiality and integrity, authentication of the server can be disabled by modifying the client’s service configuration as shown in Figure 29-15.

```
Dynamic SSLIOP_Factory Service_Object * \n   TAO_SSLIOP: make_TAO_SSLIOP_Protocol_Factory() \n   "-SSLAuthenticate NONE -SSLPrivateKey PEM:clientkey.pem \n   -SSLCertificate PEM:clientcert.pem"
```

Figure 29-15 Client-side svc.conf for confidentiality and integrity only

In this configuration, the client no longer requires authentication of the server, so the absence of the certifying authority’s certificate is of no consequence, at least so far as the client is concerned. The server’s configuration still calls for authentication of peers so the server’s environment must provide access to the certifying authority’s certificate.

## 29.7 Security Policy Controlling Application

Applications can control the message protection level and the extent to which the ORB authenticates its peers. The level of message protection is controlled by the Quality of Protection policy. The extent to which an ORB authenticates its peers is controlled by the Establish Trust policy.

The following sections:

• Describe the Quality of Protection policy.
• Describe the Establish Trust policy.
• Present code examples that demonstrate the use of these policies.

### 29.7.1 Controlling Message Protection

The `SecQOPPolicy` permits control of the Quality of Protection, and carries a value of type `Security::QOP`. The `Security` module gives the following type definition:

```
module Security { 
   // other definitions omitted
   enum QOP { 
      SecQOPNoProtection, 
      SecQOPIntegrity, 
   }
```

```
29.7 Security Policy Controlling Application

SecQOPConfidentiality,
SecQOPIntegrityAndConfidentiality

Each value represents a different level of protection:

• SecQOPNoProtection: disables all message protection mechanisms. Consequently, messages are transmitted in plaintext.

• SecQOPIntegrity: enables mechanisms that deter tampering with messages in transit and support the detection of messages that have been tampered with in some way.

• SecQOPConfidentiality: enables mechanisms that prevent a message's disclosure to all but its intended recipients.

• SecQOPIntegrityAndConfidentiality: enables both integrity and confidentiality protection mechanisms.

SSL does not support message integrity and confidentiality separately. Thus, the only practical values for SecQOPPolicy are SecQOPNoProtection and SecQOPIntegrityAndConfidentiality. The other values have no effect.

29.7.2 Controlling Peer Authentication

The SecEstablishTrustPolicy permits control of the Establish Trust policy, and carries a value of type Security::EstablishTrust. The Security module gives the type definition for EstablishTrust as follows:

module Security {
    struct EstablishTrust {
        boolean trust_in_client;
        boolean trust_in_target;
    };
    // all other definitions omitted
};

EstablishTrust::trust_in_client specifies whether or not an ORB authenticates its peers when acting in the server role.
EstablishTrust::trust_in_target specifies whether or not an ORB authenticates its peers when acting in the client role.
29.7.3 Security Policy Controlling Application Examples

We continue with the Messenger client and server to demonstrate security policies. Full source code and configuration files for this example can be found in $TAO_ROOT/DevGuideExamples/Security/PolicyControllingApp.

We use the configurations shown in Figure 29-16 and Figure 29-17 to devise a server that authenticates its peers, without regard to its role, and accepts requests via IIOP as well as SSLIOP.

```
Dynamic SSLIOP_Factory Service_Object* \
    TAO_SSLIOP:make_TAO_SSLIOP_Protocol_Factory() \
    "-SSLNoProtection -SSLAuthenticate SERVER_AND_CLIENT \ 
    -SSLPrivateKey PEM:serverkey.pem -SSLCertificate \ 
    PEM:servercert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

**Figure 29-16 Policy Controlling Application Server-side svc.conf**

The client, as configured, issues all requests via SSLIOP but does not authenticate its peers.

```
Dynamic SSLIOP_Factory Service_Object* \
    TAO_SSLIOP:make_TAO_SSLIOP_Protocol_Factory() \
    "-SSLAuthenticate NONE -SSLPrivateKey PEM:clientkey.pem \ 
    -SSLCertificate PEM:clientcert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

**Figure 29-17 Policy Controlling Application Client-side svc.conf**

Let us assume, however, that the Messenger client requires the Messenger server to be authenticated, after which it transmits messages in the clear. To accomplish this, message protection is disabled and peer authentication is enabled when operating as a client.

The client first obtains a reference to the Messenger server:

```
CORBA::Object_var obj = orb->string_to_object( "file://Messenger.ior" );
```

Next, the client constructs a `Security::SecQOPPolicy` to disable message protection:

```
Security::QOP qop = Security::SecQOPNoProtection;
CORBA::Any no_protection;
no_protection <<= qop;
```
Next, the client creates a `Security::SecEstablishTrustPolicy` to enable authentication of servers:

```cpp
def establish_trust = Security::EstablishTrust
establish_trust.trust_in_client = 0
establish_trust.trust_in_target = 1;
```

The client now assembles a policy list and uses it to create a new object reference that has the desired policies:

```cpp
corba::PolicyList policy_list (2); policy_list.length (2);
policy_list[0] = corba::Policy::_duplicate (policy.in ()); policy_list[1] = corba::Policy::_duplicate (policy2.in ());
```

Finally, the client narrows the new object reference to the appropriate type:

```cpp
corba::Object_var object =
  obj->_set_policy_overrides (policy_list, corba::SET_OVERRIDE);
```

Here is the complete client for this example:

```cpp
#include "MessengerC.h"
#include "orbsvcs/SecurityC.h"

int main( int argc, char* argv[] ) {
  try {
    corba::ORB_var orb = corba::ORB_init( argc, argv );
    corba::Object_var obj = orb->string_to_object( "file://Messenger.ior" );

    // downgrade to no message protection
    Security::QOP qop = Security::SecQOPNoProtection;
    security::Any no_protection <<= qop;
```
CORBA::Policy_var policy =
    orb->create_policy (Security::SecQOPPolicy, no_protection);
// upgrade to authenticate servers
Security::EstablishTrust establish_trust;
establish_trust.trust_in_client = 0;
establish_trust.trust_in_target = 1;
CORBA::Any want_trust;
want_trust <<= establish_trust;
CORBA::Policy_var policy2 =
    orb->create_policy (Security::SecEstablishTrustPolicy, want_trust);

// prepare to create new object reference
// having the desired policies
CORBA::PolicyList policy_list (2);
policy_list.length (2);
policy_list[0] = CORBA::Policy::_duplicate (policy.in ());
policy_list[1] = CORBA::Policy::_duplicate (policy2.in ());

// create the new object reference and
// narrow to appropriate type
CORBA::ObjectVar object =
    obj->_set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
Messenger_var messenger = Messenger::_narrow (object.in ());

CORBA::StringVar message =
    CORBA::string_dup ("Implementing security policy now!" );
messenger->send_message(
    "Chief of Security", "New Directive", message.inout() );
}
catch (CORBA::Exception& ex) {
    ex._tao_print_exception("Client: main block");
}
return 0;
}

For the next example, we change the server and client configurations to those shown in Figure 29-18 and Figure 29-19, respectively.

```
Dynamic_SSLIOP_Factory Service_Object* \
    TAO_SSLIOP_make_TAO_SSLIOP_Protocol_Factory() \ 
    "-SSLAuthenticate SERVER_AND_CLIENT \ 
    -SSLPrivateKey PEM:serverkey.pem \ 
    -SSLCertificate PEM:servercert.pem"
static Resource_Factory "/ORBProtocolFactory SSLIOP_Factory"
```

**Figure 29-18 Policy Controlling and Enforcing Server-side svc.conf**
With these changed configurations, the client must upgrade the message protection policy, instead of downgrading as demonstrated by the previous example, because the server no longer accepts requests via IIOP. The server now enforces message protection on all requests.

```c
dynamic SSLIOP_Factory Service_Object* \\
    TAO_SSLIOP:make_TAO_SSLIOP_Protocol_Factory() \\
    "-SSLNoProtection -SSLAuthenticate NONE \\
    -SSLPrivateKey PEM:clientkey.pem \\
    -SSLCertificate PEM:clientcert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Figure 29-19 Policy Controlling and Enforcing Client-side svc.conf

Here is the relevant change to the client code:

```c
Security::QOP qop = Security::SecQOPIntegrityAndConfidentiality;
CORBA::Any want_protection;
want_protection <<= qop;
CORBA::Policy_var policy = 
    orb->create_policy (Security::SecQOPPolicy, want_protection);
```

The other code remains as is. Here is the complete example:

```c
#include "MessengerC.h"
#include "orbsvcs/SecurityC.h"

int main( int argc, char* argv[] )
{
    try {
        CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );
        CORBA::Object_var obj = 
            orb->string_to_object( "file://Messenger.ior" );

        // upgrade to message protection
        Security::QOP qop = Security::SecQOPIntegrityAndConfidentiality;
        CORBA::Any want_protection;
        want_protection <<= qop;
        CORBA::Policy_var policy = 
            orb->create_policy (Security::SecQOPPolicy, want_protection);

        // upgrade to authenticate servers
        Security::EstablishTrust establish_trust;
        establish_trust.trust_in_client = 0;
        establish_trust.trust_in_target = 1;
        CORBA::Any want_trust;
        want_trust <<= establish_trust;
        CORBA::Policy_var policy2 =
```
ORB->create_policy (Security::SecEstablishTrustPolicy, want_trust);

// prepare to create new object reference
// having the desired policies
CORBA::PolicyList policy_list (2);
policy_list.length (1);
policy_list[0] = CORBA::Policy::_duplicate (policy.in ());
policy_list.length (2);
policy_list[1] = CORBA::Policy::_duplicate (policy2.in ());

// create the new object reference and
// narrow to appropriate type
CORBA::Object_var object =
    obj->_set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
Messenger_var messenger = Messenger::narrow( object.in() );
CORBA::String_var message =
    CORBA::string_dup( "Implementing security policy now!" );
messenger->send_message( "Chief of Security",
    "New Directive",
    message.inout() );
)
catch (CORBA::Exception& ex) {
    ex._tao_print_exception("Client: main block");
}
return 0;

## 29.8 Security Policy Enforcing Application

A software application can take some responsibility for ensuring compliance with security policies during its operation. A software application cannot, however, take complete responsibility because, as was discussed previously, there are many contributing factors outside a software application’s control. To contribute towards compliance with security policies, an application needs access to security context information at run time.

TAO’s implementation currently only offers a proprietary interface that provides access to security context information during secure operation invocations:

- **SSLIOP::Current** is a TAO-specific extension to the security specification. It provides access to the certificate and the certificate verification chain associated with a secure invocation’s initiator. It also
provides the means to determine whether or not the current invocation is taking place within a secure session.

The sections that follow describe this interfaces and present code examples demonstrating its use. Full source code and configuration files for this example can be found in 

Note
The forthcoming SecurityLevel3 module also offers a Current which would provide similar information as to that provided in the now-obsolete SecurityLevel1 and SecurityLevel2 interfaces.

29.8.1 SSLIOP::Current
The module SSLIOP defines the SSLIOP::Current interface and associated types. The relevant portions of this module are:

module SSLIOP {

    // portions omitted
    typedef sequence<octet> ASN_1_Cert;
    typedef sequence<ASN_1_Cert> SSL_Cert;

    local interface Current : CORBA::Current {
        exception NoContext {};

        ASN_1_Cert get_peer_certificate () raises (NoContext);
        SSL_Cert get_peer_certificate_chain () raises (NoContext);
        boolean no_context ();
    };

}

An ASN_1_Cert is a sequence of octets. Certificates are encoded using ASN1’s distinguished encoding rules (DER). The OpenSSL library includes functions for converting certificates from this format to OpenSSL’s internal format and routines for manipulating certificates once they are in the internal format.

The operation get_peer_certificate() returns the peer’s certificate in ASN1 DER format. The operation get_peer_certificate_chain() returns the sequence of certificates used to validate the initiating peer’s certificate; the chain does not include the originating principal’s certificate. These operations are available only during a secure invocation and only at the
target. Consequently, peer always refers to the operation’s initiator, i.e., to the client.

The operation no_context() indicates whether or not the current operation was invoked via a secure session. Invoking the other operations outside the scope of a secure invocation results in a SSLIOP::Current::NoContext exception.

### 29.8.2 Security Policy Enforcing Application Examples

For the first example, the Messenger server will now indicate whether or not a message was delivered via a secure invocation. The new message format is:

```
SECURE message from: Chief of Security
Subject:             New Directive
Message:             Implementing security policy now!
```

We use SSLIOP::Current::no_context() to determine the nature of the invocation, secure or not secure. Several changes are necessary to provide this capability.

First, the Messenger_i servant needs a reference to SSLIOP::Current that is accessible during the invocation. For that purpose, we add an SSLIOP::Current_var attribute to the Messenger server object:

```c++
#include <orbsvcs/SSLIOPC.h>
#include "MessengerS.h"

class Messenger_i : public virtual POA_Messenger {
public:
    Messenger_i (       
        SSLIOP::Current_ptr sssiop_current 
    );

    virtual ~Messenger_i (void);

    virtual CORBA::Boolean send_message (       
        const char* user_name,       
        const char* subject,       
        char*& message 
    ) throw( CORBA::SystemException );

protected:
    SSLIOP::Current_var sssiop_current_;
};
```
A reference to SSLIOP::Current is passed to the Messenger_i servant upon construction and stored in sslip_current_:

```
#include "Messenger_i.h"
#include <iostream>

Messenger_i::Messenger_i {
    SSLIOP::Current_ptr sslip_current
} : sslip_current_(SSLIOP::Current::_duplicate(sslip_current))
{
}
```

Now the Messenger server’s main function must obtain a reference to SSLIOP::Current to pass to the Messenger_i servant’s constructor. A reference to SSLIOP::Current is obtained via the interface CORBA::ORB::resolve_initial_references(). “SSLIOPCurrent” is the name of the SSLIOP::Current object.

```
obj = orb->resolve_initial_references ("SSLIOPCurrent");
SSLIOP::Current_var sslip_current =
    SSLIOP::Current::_narrow (obj.in ());
Messenger_i messenger_servant(sslip_current.in());
```

Finally, the message output function uses the boolean operation SSLIOP::Current::no_context() to select a message format:

```
CORBA::Boolean Messenger_i::send_message {
    const char* user_name,
    const char* subject,
    char*& message
}
throw ( CORBA::SystemException )
{
    if (sslip_current_->no_context())
        std::cout << "Message from: " << user_name << std::endl;
    else
        std::cout << "SECURE message from: " << user_name << std::endl;

    std::cout << "Subject: " << subject << std::endl;
    std::cout << "Message: " << message << std::endl;
    std::cout << std::endl;
    return 1;
}
```
For the second example, we introduce a new operation called `shutdown()` that stops the Messenger server. However, we only allow authenticated users to shut down the Messenger service. Therefore, make sure that all invocations of `shutdown()` occur within a secure session by examining the return value of `SSLIOP::Current::no_context()`.

Here is the new interface definition:

```c++
interface Messenger {
    boolean send_message ( in    string user_name,
                           in    string subject,
                           inout string message );

    void shutdown ( in string user_name );
};
```

The ORB is shut down via the operation `CORBA::ORB::shutdown()`. The Messenger_i servant now needs a reference to the ORB as well as a reference to `SSLIOP::Current`.

Here is the updated Messenger_i servant header file:

```c++
#include <orbsvcs/SSLIOPC.h>
#include "MessengerS.h"

class  Messenger_i : public virtual POA_Messenger {
public:
    Messenger_i ( CORBA::ORB_ptr orb,
                  SSLIOP::Current_ptr ssliop_current );

    virtual ~Messenger_i (void);

    virtual CORBA::Boolean send_message ( const char* user_name,
                                          const char* subject,
                                          char*& message )
        throw ( CORBA::SystemException );

    virtual void shutdown ( const char* user_name )
        throw ( CORBA::SystemException );

protected:
}
```
Now the implementation changes. First the constructor:

```cpp
Messenger_i::Messenger_i (CORBA::ORB_ptr orb,
                         SSLIOP::Current_ptr ssliop_current
) :
    orb_(CORBA::ORB::_duplicate(orb)),
    ssliop_current_(SSLIOP::Current::_duplicate(ssliop_current))
{};
```

And the `shutdown()` operation:

```cpp
void Messenger_i::shutdown (const char* user_name) throw (CORBA::SystemException)
{
    if ( ! (ssliop_current_->no_context()) ) {
        std::cout << "Shutdown command from: " << user_name << std::endl;
        std::cout << "Status: User authenticated." << std::endl;
        std::cout << "Action: Sever shutdown in progress..." << std::endl;
        orb_->shutdown (0);
    }
    else {
        std::cout << "Shutdown command from: " << user_name << std::endl;
        std::cout << "Status: User *NOT* authenticated." << std::endl;
        std::cout << "Action: Ignored." << std::endl;
    }
}
```

After the servant calls `CORBA::ORB::shutdown()`, `CORBA::ORB::run()` returns in `main`. We announce the service is shut down and clean up before exiting. Here are the changes to the main function:

```cpp
#include "Messenger_i.h"
#include <iostream>

int
```
main( int argc, char* argv[] ) {
    try {
        // Not shown:
        // Init the ORB,
        // get and activate the Root POA

        obj =
            orb->resolve_initial_references("SSLIOPCurrent");

        SSLIOP::Current_var ssliop_current =
        SSLIOP::Current::_narrow(obj.in());

        Messenger_i messenger_servant(orb.in(), ssliop_current.in());

        // Not shown:
        // activate the messenger servant,
        // create and export a Messenger
        // object reference

        orb->run();
        poa->destroy(1, 1);
        orb->destroy();

        std::cout << "Messenger Server is shut down!"
            << std::endl;
        std::cout << std::endl;
    }

    // Not shown:
    // exception handling

    return 0;
}

The Messenger service still handles message requests submitted in the clear
by unauthenticated clients, but it will not process a shutdown request unless it
is submitted via a secure session:

Message from: Chief of Security
Subject: New Directive
Message: Terminating Messenger service!

Shutdown command from: Chief of Security
Status: User *NOT* authenticated.
Action: Ignored.
Here is the complete implementation for `Messenger::shutdown()`:

```cpp
void Messenger_i::shutdown (const char* user_name)
    throw (CORBA::SystemException)
{
    if (! (ssliop_current_->no_context())) {
        // client has been authenticated

        std::cout << "Shutdown command from: " << user_name << std::endl;
        std::cout << "Status:                User authenticated." << std::endl;
        std::cout << "Action:                Sever shutdown in progress..." << std::endl;
        std::cout << std::endl;
        orb_->shutdown (0);
    } else {
        // requester has not been authenticated

        std::cout << "Shutdown command from: " << user_name << std::endl;
        std::cout << "Status:                User *NOT* authenticated." << std::endl;
        std::cout << "Action:                Ignored." << std::endl;
        std::cout << std::endl;
    }
}
```

Here is the complete Messenger server’s main function:

```cpp
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main( int argc, char* argv[] )
{
    try {
        CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );
        CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa =
            PortableServer::POA::_narrow( obj.in() );
        PortableServer::POAManager_var mgr =
            poa->the_POAManager();
        mgr->activate();

        obj = orb->resolve_initial_references("SSLIOPCurrent");
        SSLIOP::Current_var ssliop_current =
            SSLIOP::Current::_narrow (obj.in ());
```
Messenger_i messenger_servant(
    orb.in(),
    security_current.in(),
    sslcio_current.in());

PortableServer::ObjectID_var oid =
    poa->activate_object(&messenger_servant);
CORBA::Object_var messenger_obj =
    poa->id_to_reference( oid.in() );
CORBA::String_var str =
    orb->object_to_string( messenger_obj.in() );
std::ofstream iorFile("Messenger.ior");
iorFile << str.in() << std::endl;
iorFile.close();

std::cout << "IOR written to file Messenger.ior" << std::endl;

orb->run();
poa->destroy (1, 1);
orb->destroy ();

std::cout << "Messenger Server is shut down!"
    << std::endl
    << std::endl;
}
catch( const CORBA::Exception& ex ) {
    ex._tao_print_exception("Server Error: main block");
    return 1;
}

return 0;

29.9 SSLIOOP Factory Options

The remainder of this chapter describes the individual options interpreted by the default SSLIOOP factory. These options are applied to the default SSLIOOP factory by the service configurator as described in 29.6.2.

29.9.1 verbose

Description This switch turns on debugging messages during loading of the SSLIOOP protocol factory.
Usage  The default behavior is equivalent to not specifying this option, i.e., no debugging messages get printed.

Impact  Specifying this option prints messages equivalent to passing -ORBDebugLevel 1 on the command line. Note that you cannot use -ORBDebugLevel to get the same effect, because protocol factories get loaded before -ORBDebugLevel gets processed.

29.9.2 SSLNoProtection

Description  This switch controls whether or not SSLIOP is used to submit and receive requests when the SSLIOP protocol factory is loaded.

Usage  The default behavior of the SSLIOP protocol factory is to only submit and receive requests made using SSLIOP.

Impact  Specifying this switch allows insecure IIOP requests to be sent and received by ORB clients and servants.

See Also  29.6.2, 29.7.3

Example  dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP: make_TAO_SSLIOP_Protocol_Factory() \ "-SSLNoProtection"

29.9.3 SSLCertificate

Description  This option specifies the type of certificate, either PEM or ANS1, and the pathname of the certificate used by the principal.

See Also  29.6.2, 29.6.3

Example  dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP: make_TAO_SSLIOP_Protocol_Factory() \ "-SSLCertificate PEM:client_cert.pem"

29.9.4 SSLDHparams

Description  This option specifies the type of encoding, either PEM or ANS1, and the pathname of the Diffie-Hellman cipher parameters. Note that SSLIOP only recognizes the PEM encoding, and gives an error if you specify ASN1

See Also  29.6.2, 29.6.3

Example  dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP: make_TAO_SSLIOP_Protocol_Factory() \ 
"-SSLCertificate PEM:client_cert.pem \\
-SSLDHPParams PEM:dhparams.pem"

29.9.5 SSLPrivateKey

Description This option specifies the type of private key, either PEM or ANS1, and the pathname of the private key used by the principal.

See Also 29.6.2, 29.6.3

Example dynamic SSLIOP_Factory Service_Object* 
TAO_SSLIOP: make_TAO_SSLIOP_Protocol_Factory() \\
"-SSLPrivateKey PEM:client_key.pem"

29.9.6 SSLAuthenticate

Values for SSLAuthenticate Option

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE (default)</td>
<td>Disable peer authentication.</td>
</tr>
<tr>
<td>SERVER</td>
<td>Client process will authenticate peers.</td>
</tr>
<tr>
<td>CLIENT</td>
<td>Server process will authenticate peers. The SSL protocol causes this setting to be equivalent to SERVER_AND_CLIENT.</td>
</tr>
<tr>
<td>SERVER_AND_CLIENT</td>
<td>Client and Server processes will authenticate peers.</td>
</tr>
</tbody>
</table>

Description This option specifies the level of peer authentication conducted as secure sessions are established between a client and a server.

Impact This option impacts the authentication required to participate in a secure session.

See Also 29.6.2, 29.6.3, 29.7.3

Example dynamic SSLIOP_Factory Service_Object* 
TAO_SSLIOP: make_TAO_SSLIOP_Protocol_Factory() \\
"-SSLAuthenticate SERVER_AND_CLIENT"

29.9.7 SSLAcceptTimeout

Description This option specifies the amount of time, as a floating point value, that SSLIOP will wait to complete the SSL handshake. The value is in seconds, with any remainder treated as microseconds.

Impact The SSL handshake includes a TCP handshake, so the timeout value should take this the TCP handshake duration into account.
See Also 29.6.2, 29.6.3, 29.3.3.1

Example
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLPrivateKey PEM:client_key.pem"
CHAPTER 30

Implementation Repository

30.1 Introduction

The TAO Implementation Repository (ImR) is a service which allows indirect invocation of CORBA operations. Instead of connecting to a server, clients initially connect to the ImR. The ImR will then optionally start the appropriate server if the server is not running, and then provide the address of the server to the client so that subsequent requests will go to this server.

Indirectly binding with the server in this manner can be useful in minimizing the changes needed to accommodate server or object migration, allow for automatic server start-up, and help in load balancing.

Chapter 3 demonstrated how a client uses an object reference to establish a connection with a server. Generally, when a server receives a request, it has to associate the request with one of its servants for processing. The process of a server opening a connection and associating the object reference with a servant is known as binding. TAO servers support both direct and indirect binding.
30.1.1 Direct Binding
When using direct binding, the server address and an object key are embedded in an object’s IOR, which in URL form might look something like:

corbaloc:iiop:1.2@host:port/object_key

The first time a client invokes an operation on the object, the client ORB extracts the address from this IOR and uses it to connect to a server. The client ORB uses this connection to send requests to the server, and the server ORB uses the object key to locate a servant for the request.

30.1.2 Indirect Binding
When using indirect binding, the ImR address is substituted for the actual server address in the IOR. When the ImR receives a client request it extracts the POA name from the object key, and uses it to find a matching registered server IOR. The ImR then responds with a LOCATION_FORWARD message which instructs the client to repeat the request using the new IOR. Subsequent client requests using the same object reference continue to use the established server connection, and the ImR is not involved.

**Note**
It is possible for a client to create another connection. For example, upon receiving a COMM_FAILURE exception, a client may decide to retry by invoking the operation again. The ImR will again forward the client to the registered server if possible.

Throughout this chapter, a POA that uses a PERSISTENT lifespan policy and a USER_ID object id assignment policy is referred to as a persistent POA, and an object that is registered with a persistent POA is referred to as a persistent object. In TAO all persistent POAs are registered with the ImR upon POA creation if the -ORBUseImR 1 option is specified at ORB initialization. The TAO_USE_IMR environment variable controls the default value for this setting. Furthermore, any object references use indirect binding.

Keep in mind that there is a performance penalty for the first indirect request. If there are a large number of client requests the interaction with the ImR may become a bottleneck in the system.

Frequently this documentation uses the terms POA and server interchangeably, because the ImR treats each registered POA as a separate
server. However, each server is capable of hosting multiple persistent POAs, each of which may contain multiple indirect object references.

The ImR supports the ability to launch server processes on demand. Without the ImR all servers must be running prior to any client request. This can result in performance problems, especially for large systems, since servers consume system resources even in their idle states.

The CORBA specification deliberately avoids standardizing the ImR behavior, mentioning only that indirect binding should be supported, and providing the LOCATION_FORWARD mechanism. This is sufficient to provide portability from the client perspective, but any server/ImR interaction described in this chapter is specific to TAO.

**Note** More information about binding and the ImR can be found in Chapter 14 of Advanced CORBA Programming with C++. TAO’s ImR implementation was originally based on the one described in that book.

### 30.2 The Operation of the ImR

ImR requires that an IOR it works with live beyond the server that created the IOR. Therefore, the IOR must be created using a POA with the PERSISTENT life span policy. The persistent POA registers itself with the ImR when it is created if the ORB was initialized using -ORBUseIMR 1 (or TAO_USE_IMR=1). If a server has more than one persistent POA, each is registered as a separate server within the ImR. For each persistent POA, the following information is registered:

- POA name
- Address
- ServerObject

The TAO-specific ServerObject is a simple object which is activated in the Root POA, and supports shutdown() and ping() operations.

The address is actually stored as the first part of the ServerObject IOR in URL format. (e.g., corbaloc:iiop:1.2@127.0.0.1:8888/)

The server must be able to find the ImR to register this information. (This happens automatically if an Activator, discussed in 30.4.1, is used.) This is
achieve by passing a -ORBInitRef ImplRepoService=... option, set the ImplRepoServiceIOR environment variable, or use the IOR multicast feature to allow servers, Activators, and the ImR utility to find the ImR.

Additionally the ImR maintains the following information about each registered server:

• Activation mode (discussed in 30.5).
• Start-up command
• Environment variables
• Working directory
• Start retry limit

Most of the above information is registered using the tao_imr utility, or read from persistent storage when the ImR is started. The start-up command, working directory, and environment variables are optional. However, a start-up command is required for the ImR to start a server process. The ImR uses the Start Retry Limit to automatically reattempt starting a server if it fails to start the first time. If this limit is exceeded then the ImR does not attempt to start the server again, and a status indicating the ImR is locked is sent to standard output until the start-up count is reset using the tao_imr utility, or until the server is started.

If the ImR does not already contain a record for a particular POA name when a server POA is registered, then one is created with NORMAL activation mode, and no start-up options. This can be modified later using the tao_imr utility if necessary.

Because the ImR is indexed by persistent POA name, no two servers using the same ImR may use the same persistent POA name.

If a server dies unexpectedly while communicating with a client, then the client may get a COMM_FAILURE exception. If the client simply re-attempts the operation, then it is once again directed through the ImR, which results in a TRANSIENT exception unless start-up information is registered for the server.

---

Note: The ping() operation described in 30.4 is disabled if no start-up information is registered, because the ImR would only return a TRANSIENT exception anyway.
30.2.1 Basic Indirection

When a server is already running, the steps involved in a client sending requests to a server using the ImR are as follows:

1. Client sends a request to ImR.
2. ImR looks up the server, and sees that it is running.
3. ImR sends LOCATION_FORWARD reply to Client.
4. Client sends request to Server.
5. Server sends reply to Client.

Note The client will continue to use the new address for future invocations until a COMM_FAILURE or TRANSIENT exception occurs, after which the client will revert to the address of the ImR until forwarded again.

30.3 Basic Indirection Example

The following example illustrates how to achieve indirect binding without the ability to start the servers on demand. The source code for this example can be found in $TAO_ROOT/DevGuideExamples/ImplRepo/Basic.

30.3.1 Create the Server

We use a slightly modified version of our typical MessengerServer for this example. The only change we have to make is to use a persistent POA. The Messenger_i.h and Messenger_i.cpp files remain unchanged.

30.3.1.1 MessengerServer.cpp

```c++
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

PortableServer::POA_ptr createPersistentPOA ( 
    PortableServer::POA_ptr root_poa, const char* poa_name) 
{

    CORBA::PolicyList policies(2);
    policies.length(2);

    policies[0] = root_poa->create_lifespan_policy ( 
```
PortableServer::PERSISTENT);
policies[1] = root_poa->create_id_assignment_policy (  PortableServer::USER_ID);

PortableServer::POAManager_var mgr = root_poa->the_POAManager();
PortableServer::POA_var poa = root_poa->create_POA (  poa_name, mgr.in(), policies);

policies[0]->destroy();
policies[1]->destroy();

return poa._retn();
}

void writeIORFile(const char* ior)
{
    std::ofstream out("Messenger.ior");
    out << ior << std::endl;
}

int main(int argc, char* argv[])
{
    try {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var root_poa = PortableServer::POA::narrow(obj.in());
        PortableServer::POAManager_var mgr = root_poa->the_POAManager();

        const char* poa_name = "MessengerService";
        PortableServer::POA_var poa = createPersistentPOA(root_poa.in(), poa_name);

        Messenger_i servant;
        CORBA::ObjectId_var object_id = PortableServer::string_to_ObjectId("object");
        poa->activate_object_with_id(object_id.in(), &servant);

        obj = poa->id_to_reference(object_id.in());
        CORBA::String_var ior = orb->object_to_string(obj.in());
        writeIORFile(ior.in());

        mgr->activate();

        std::cout << "Messenger server ready." << std::endl;

        orb->run();
    }
    catch (std::exception& e) {
        std::cout << e.what() << std::endl;
    }
}
30.3 Basic Indirection Example

```cpp
std::cout << "Messenger server shutting down." << std::endl;

root_poa->destroy(1,1);
orb->destroy();

return 0;
}  
catch (CORBA::Exception & ex) {
    std::cerr << "Server main() Caught Exception " << ex << std::endl;
}
return 1;
}

30.3.1.2 MessengerClient.cpp
#include "MessengerC.h"
#include <iostream>

int main(int argc, char* argv[]) {
    try {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");

        Messenger_var messenger = Messenger::_narrow(obj.in());
        if (is_nil(messenger.in())) {
            std::cerr << "Unable to get a Messenger reference." << std::endl;
            return 1;
        }

        CORBA::String_var message = CORBA::string_dup("Hello!");
        messenger->send_message("TAO User", "TAO Test", message.inout());
        std::cout << "message was sent" << std::endl;
        std::cout << "Reply was : " << message.in() << std::endl;

        return 0;
    }
    catch (CORBA::Exception & ex) {
        std::cerr << "Client main() Caught Exception: " << ex << std::endl;
    }
    return 1;
}

30.3.2 Run the Example
Now that we have a modified version of our server we can start several
command windows, and demonstrate how it works.
1. Start the ImplRepo_Service

For simplicity, we set the ImplRepoServiceIOR environment variable, which allows applications, such as our server and the tao_imr utility, to find the ImR without the need to supply lengthy -ORBInitRef options on the command line. Alternatively we could start the ImR with -m to allow clients to find it using multicast service discovery.

```bash
$ ImplRepoServiceIOR=corbaloc::localhost:8888/ImR; export ImplRepoServiceIOR
$ cd $TAO_ROOT/orbsvcs/ImplRepo_Service
$ ./ImplRepo_Service –ORBListenEndpoints iiop://:8888
```

Implementation Repository: Running
- Ping Interval : 10000ms
- Startup Timeout : 60s
- Persistence : Disabled
- Multicast : Disabled
- Debug : 1
- Locked : False

2. Start the Server

We start the server, passing -ORBUseIMR 1, so that it registers itself with the ImR. Alternatively, we could have set the TAO_USE_IMR environment variable to make this the default.

```bash
$ MessengerServer –ORBUseIMR 1
Messenger server ready.
```

The ImR output shows that our server was automatically registered.

```bash
ImR: Server MessengerService is running at corbaloc:iiop:1.2@192.168.1.10:1323/
ImR: Auto adding NORMAL server:<MessengerService>
```

3. Run the Client

```bash
$MessengerClient
message was sent
Reply was : A reply.
```

The ImR output shows that it forwarded the client to our server.

```bash
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1323/...NUP...MessengerService...object>
```

(Some parts of the IOR above replaced with ellipsis for brevity)
The server output shows a successful invocation.

Message from: TAO User
Subject: TAO Test
Message: Hello!

30.4 Server Start-up

The exact conditions under which the ImR decides to start a server are determined by the Activation Mode described in 30.4.1, but typically the ImR starts a server if it is not registered as running, or if the server cannot be pinged successfully. Once the server has been started, the ImR waits for the server to register its running information, which includes a ServerObject and a partial IOR. Multiple simultaneous client invocations are supported, and each blocks, waiting on the server to register this information. Each persistent POA in the server is treated as a separate server registration within the ImR, and currently this registration happens as soon as the POA is created.

Note

In the future this registration may be delayed until the POA Manager is activated. This would prevent problems with servers notifying the ImR before they are actually ready to handle requests.

Once the server registers its running information, the ImR wakes one of the waiting client operations, and uses it to ping the server to ensure that it is really running. The result of the ping determines whether the server is running, not running, or is in an indeterminate state.

• Running

If the server is running, or if the start-up retry count has been exceeded, then all clients are awakened and forwarded to the server. By connecting the clients to the server if the retry count was exceeded the clients can get the appropriate error status directly from the server.

• Not Running

If the server is not running, then the whole start-up process is repeated if a start-up retry count is configured for the server.

• Indeterminate
If the status cannot be determined, the ping repeats a fixed number of times with an increasing delay between subsequent attempts.

To more efficiently handle multiple client requests, the `ping()` operation has a defined interval that is passed to the ImR at start-up. If a ping has successfully completed within the specified interval, then the server is assumed to be running.

The `ping()` operation has a very short timeout configured, and a timeout is considered proof that the server is running. A well-written server should avoid activating the POA Manager until the server is actually ready to handle requests.

In some instances, a server may not register its running status with the ImR within the allowed start-up time. (See the `-t` command line option for the ImR.) This is treated as a start-up failure, and start-up may be retried as described previously.

If the start-up retry count is exceeded then a server becomes locked, and the ImR will not attempt to start that server again. Instead it immediately returns a `TRANSIENT` exception to any waiting clients. You can reset the start-up retry count using the `tao_imr` utility.

If a server dies unexpectedly while communicating with a client, then the client may get a `COMM_FAILURE` exception. If the client simply reattempts the operation, then it is once again directed through the ImR, which may allow the ImR to restart the server. However, if the server was pinged successfully within the configured ping interval, then the client may be redirected to the dead server, and receive a `TRANSIENT`.

### 30.4.1 Using An Activator

The ImR has the ability to start servers using a separate process called the ImR Activator. The Activator is a simple program that runs on the same machine as a server, and is capable of spawning processes. The ImR sends the command line, working directory, and environment variables to the Activator which uses this information to start the server. The Activator returns immediately, and does not maintain any kind of ownership of the launched processes.
On UNIX and UNIX-like platforms the Activator can monitor the launched processes for cleanup purposes, and can optionally notify the ImR when processes die.

The Activator automatically sets the ImplRepoServiceIOR and TAO_USE IMR environment variables for all spawned processes. This means that servers started by the Activator do not need to pass -ORBUseIMR 1, nor do they have to specify an -ORBInitRef ImplRepoService=... option.

A server started by the Activator typically responds to the first ping with a TRANSIENT exception indicating that the POA is in a holding state, because the POA Manager has not been activated. This causes the ImR to retry the ping. This second ping typically times out, which is treated as a successful ping. This is because in TAO 1.4a you do not get a second TRANSIENT exception unless orb->run() has been called, or another request is sent from the server.

The steps involved in a client being able to connect to a non-running server using an ImR and Activator are given below.

The first task is to activate the server using the Activator:

1. Client sends a request to ImR.
2. ImR looks up the server, and sees it is not running.
3. ImR sends a start request to ImR Activator.
4. Activator starts the Server.
5. Activator sends a start reply to ImR.
6. Server sends a running request to ImR.
7. ImR sends a ping() request to Server.
8. Server sends a ping() reply to ImR.

With the server waiting for requests the ImR can then inform the client to send requests to the server:

1. ImR sends a Location Forward reply to client.
2. ImR sends a running reply to Server.
3. Client sends a request to Server.
4. Server sends a reply to Client.
Future versions of the ImR may change the POA implementation so that the server registers its running information when the POA Manager is activated instead of when the POA is created. This would eliminate steps #7 and #8 above, because registration of running information will be treated as a successful ping.

### 30.4.2 Activator Example

The basic indirection example was enough to demonstrate the primary usage of the ImR, but often the ImR is also used to automatically start servers as needed. This example illustrates the use of the ImR and ImR Activator to start servers on demand. For this to work, an ImR Activator must be started and running on the same host as the server. We must also register a valid command line for our server using the `tao_imr` utility.

Our source code remains unchanged from the example in 30.3. However, the script to run the example is different and can be found in `$TAO_ROOT/DevGuideExamples/ImplRepo/Activator`.

#### 30.4.2.1 Run the Example

1. **Start the ImplRepo_Service**

   $ ImplRepoServiceIOR=corbaloc::localhost:8888/ImR; export ImplRepoServiceIOR
   $ cd $TAO_ROOT/orbsvcs/ImplRepo_Service
   $ ./ImplRepo_Service -ORBListenEndpoints iiop://:8888
   Implementation Repository: Running
   Ping Interval : 10000ms
   Startup Timeout : 60s
   Persistence : Disabled
   Multicast : Disabled
   Debug : 1
   Locked : False

2. **Start an Activator**

   The activator must be started on the same host on which we want to run the server.

   $ $TAO_ROOT/orbsvcs/ImplRepo_Service/ImR_Activator
   ImR Activator: Starting MYHOST
   ImR Activator: Registered with ImR.

   The ImR output shows that our activator registered correctly.
30.4 Server Start-up

ImR: Activator registered for MYHOST.

3. Register the server

We assume $ACE_ROOT/bin, where the tao_imr utility is located, is in your PATH. We accept the defaults for the Activator name, and other start-up options. The server’s command line does not require the -ORBUseIMR 1 or -ORBInitRef ImplRepoService=corbaloc::localhost:8888/ImR options, because these will be supplied automatically by the Activator.

$ tao_imr add MessengerService -c "MessengerServer"
Successfully updated <MessengerService>.

The ImR output shows that we added the server.

ImR: Add/Update server <MessengerService>

4. Run the Client

In our example, the client relies on the existence of a file named messenger.ior that contains the Messenger object’s IOR. You must either run the server once to generate the file, or use the file created in the basic indirection example.

$ MessengerClient
message was sent
Reply was : A reply.

The ImR output shows that it forwarded the client to our server.

ImR: Starting server <MessengerService>. Attempt 1/1.
ImR: Waiting for <MessengerService> to start...
ImR: Server MessengerService is running at corbaloc:iiop:1.2@192.168.1.10:1323/.
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1323/_NUF_MessengerService...object>

(Some parts of the IOR above replaced with ellipsis for brevity.)

The ImR Activator output shows that it successfully started the server.

ImR Activator: Successfully started <MessengerService>.

The server output shows a successful invocation.
30.5 Activation Modes

The ImR supports registering servers with one of four different activation modes. These affect how a server is started, and have no meaning if a server is not startable. For a server to be startable, it must have a registered command line and Activator name, and the corresponding Activator must be registered and running.

**Note** If a server registers itself automatically then no Activator or command line will be associated.

You must configure the activation mode using the `tao_imr add` or `update` command. Alternatively, you may stop the ImR, edit the XML or Windows registry persistent data manually, then start the ImR to initiate the changes. The ImR has the ability to start a server on demand, at ImR start-up, or in response to commands from the `tao_imr` utility.

Valid activation modes are:

- **normal**
  The common usage of the ImR is to have it automatically start any servers as needed for incoming client requests. This mode also allows servers to be manually started using the `tao_imr activate` command.

- **auto_start**
  This behaves exactly like normal mode, except that the ImR attempts to start the server as the ImR itself is started. You can also use the `tao_imr autostart` command to manually start these servers.

- **manual**
  This prevents the server from starting automatically as the result of an incoming client request. You must use the `tao_imr activate` command to start the server, or start the server manually using some other mechanism.
30.6 Using the ImR and the IOR Table

- **per_client**

  The name of this mode can be misleading, because the ImR does not actually keep track of which clients have attempted to use a server. Instead this simply means that a new server will be spawned for each incoming request to the ImR. Once a client has been forwarded by the ImR to an actual server, the client will continue to use this connection for future communications until a **TRANSIENT** or **COMM_FAILURE** exception occurs. In this case, the client will make a connection back to the ImR, and the process will repeat.

  *Note*  
  *It is possible for a client to make a second connection using the indirect object reference, and this will cause the ImR to launch another server. For this reason, **per_client** activation should be used with care.*

  *Note*  
  *See 30.6 for special steps that must be taken to use **per_client** activation with an IOR Table.*

30.6 Using the ImR and the IOR Table

The IOR Table described in Chapter 14 can also be used in servers that use the ImR. Recall that using the IOR Table costs an additional level of indirection with most servers. The first time a client accesses the server, it is forwarded by the IOR Table back to the server itself using the bound IOR. When combined with the ImR, the result is two levels of indirection. First the client is forwarded by the ImR to the server, and then the server forwards the client back to itself using the IOR Table.

Servers which use the ImR always change all object references associated with persistent POAs to point to the ImR (i.e., indirect binding). This needs to be done in case the server later terminates and needs to be restarted. In this case the IOR registered in the IOR Table points at the ImR, which causes another level of indirection. To avoid this extra cost, you can use a TAO specific function to create a direct bound object reference for use in the IOR Table.

```c++
TAO_Root_POA* tpoa = dynamic_cast<TAO_Root_POA*>(poa.in());
```
CORBA::Object_var obj = tpoa->id_to_reference(id, false);
CORBA::String_var ior = orb->object_to_string(obj.in());

You **must** use this code for **per_client** activation, or two servers will be started for each client invocation.

**Note**  
*The mechanism for creating a direct bound object reference is subject to change in subsequent versions of TAO, and was not available prior to TAO 1.4a.*

The `tao_imr` utility can be used to generate IORs for any registered server even if the server is not running. It simply constructs a `corbaloc` object URL using a simple object key passed as an argument. (e.g., `corbaloc:iiop:1.2@127.0.0.1:8888/MyPOA`). For this to work, you must register the simple key with the IOR Table in your server.

In TAO 1.4a, you can now bind multiple IORs to simple object keys for each persistent POA. Simply use a forward slash `'/` to separate the POA name from the object key.

```
iorTable->bind("MessengerService/Messenger1", ior1);
iorTable->bind("MessengerService/Messenger2", ior2);
```

This is accessed using an object URL such as:

`corbaloc:iiop:1.2@127.0.0.1:8888/MessengerService/Messenger1`

This object URL can be generated using the following `tao_imr` command:

```
tao_imr ior MessengerService/Messenger1
```

For the best performance, you should avoid use of the IOR Table due to the extra level of indirection it introduces. To ensure that clients always access your server in the most efficient way possible, do not register anything with the IOR Table. This forces clients to use IORs, or object URLs with full object keys.

To ensure that your object references are always valid, you must be sure to start the server on a consistent endpoint. For example:

```
-ORBListenEndpoints iiop://:9999
```
To make the IORs more human-readable, you may also want to use the -ORBObjRefStyle URL option (see 19.8.24).

### 30.6.1 The Steps in Using an IOR Table

When a client sends a request the ImR that will become fulfilled by a server using an IOR Table that is already running, the following steps are taken to allow the client to send a request to the server:

1. Client sends a request to ImR.
2. ImR looks up the server, and sees that it is running.
3. ImR sends a ping() request to Server.
4. Server sends a ping() reply to ImR.
5. ImR sends a Location Forward reply to Client.
6. Client sends a request to Server.
7. Server sends a Location Forward reply to Client.
8. Client sends a request to Server.
9. Server sends a reply to Client.

### 30.7 ImR and IOR Table Example

We modify our first example to show how to support an IOR Table. We also show how to register more than one object in a persistent POA, and we expose these objects using simple URLs. The source code for this example can be found in `$TAO_ROOT/DevGuideExamples/ImplRepo/IORTable`.

#### 30.7.1 Create the Server

We modify our existing `MessengerServer.cpp` to support our new requirements.

#### 30.7.1.1 MessengerServer.cpp

```cpp
#include "Messenger_i.h"
#include <tao/PortableServer/POA.h>
#include <tao/IORTable/IORTable.h>
#include <iostream>
#include <fstream>
```
PortableServer::POA_ptr createPersistentPOA(
    PortableServer::POA_ptr root_poa, const char* poa_name)
{

    CORBA::PolicyList policies(2);
    policies.length(2);

    policies[0] = root_poa->create_lifespan_policy (PortableServer::PERSISTENT);
    policies[1] = root_poa->create_id_assignment_policy (PortableServer::USER_ID);

    PortableServer::POAManager_var mgr = root_poa->the_POAManager();
    PortableServer::POA_var poa = root_poa->create_POA (poa_name, mgr.in(), policies);

    policies[0]->destroy();
    policies[1]->destroy();

    return poa._retn();
}

void writeIORFile(const char* ior, const char* name)
{
    std::ofstream out(name);
    out << ior << std::endl;
}

int main (int argc, char *argv[])
{
    try {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var root_poa = PortableServer::POA::_narrow(obj.in());

        PortableServer::POAManager_var mgr = root_poa->the_POAManager();
        const char* poa_name = "MessengerService";

        PortableServer::POA_var poa = createPersistentPOA (root_poa.in(), poa_name);
        Messenger_i servant1, servant2;

        CORBA::ObjectId_var id1 = PortableServer::string_to_ObjectId("Messenger1");
        poa->activate_object_with_id(id1.in(), &servant1);
        CORBA::ObjectId_var id2 = PortableServer::string_to_ObjectId("Messenger2");
        poa->activate_object_with_id(id2.in(), &servant2);
30.7 ImR and IOR Table Example

```cpp
obj = poa->id_to_reference(id1.in());
CORBA::String_var ior1 = orb->object_to_string(obj.in());
obj = poa->id_to_reference(id2.in());
CORBA::String_var ior2 = orb->object_to_string(obj.in());

TAO_Root_POA* tpoa = dynamic_cast<TAO_Root_POA*>(poa.in());
obj = tpoa->id_to_reference(id1.in(), false);
CORBA::String_var direct_ior1 = orb->object_to_string(obj.in());

obj = orb->resolve_initial_references("IORTable");
IORTable::Table_var ior_table = IORTable::Table::narrow(obj.in());
ior_table->bind("MessengerService/Messenger1", direct_ior1);
// Bind using an indirect reference.
ior_table->bind("MessengerService/Messenger2", ior2);

writeIORFile(ior1.in(), "Messenger1.ior");
writeIORFile(ior2.in(), "Messenger2.ior");

mgr->activate();
std::cout << "Messenger server ready." << std::endl;

orb->run();
std::cout << "Messenger server shutting down." << std::endl;

root_poa->destroy(1,1);
orb->destroy();

return 0;
}
```
ACE_CString ior = "file://";
ior += argv[1];

CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

CORBA::Object_var obj = orb->string_to_object(ior.c_str());

Messenger_var messenger = Messenger::narrow(obj.in());
if (CORBA::is_nil(messenger.in())) {
    std::cerr << "Unable to get a Messenger reference." << std::endl;
    return 1;
}

CORBA::String_var message = CORBA::string_dup("Hello!");
messenger->send_message("TAO User", "TAO Test", message.inout());
std::cout << "message was sent" << std::endl;
std::cout << "Reply was : " << message.in() << std::endl;
return 0;
}

catch (CORBA::Exception& ex) {
    std::cerr << "Client main() Caught Exception: " << ex << std::endl;
}
return 1;

30.7.2 Run the Example
Now that we have a modified version of our server we can start several command windows, and demonstrate how it works.

1. Start the ImplRepo_Service
For simplicity, we set the ImplRepoServiceIOR environment variable, which allows applications, such as our server and the tao_imr utility, to find the ImR without the need to supply lengthy -ORBInitRef options on the command line.

    $ ImplRepoServiceIOR=corbaloc::localhost:8888/ImR; export ImplRepoServiceIOR
    $ cd $TAO_ROOT/orbsvcs/ImplRepo_Service
    $ ./ImplRepo_Service -ORBListenEndpoints iiop://:8888
Implementation Repository: Running
    Ping Interval : 10000ms
    Startup Timeout : 60s
    Persistence : Disabled
    Multicast : Disabled
    Debug : 1
2. Start the Server

We start the server, passing `–ORBUseIMR 1`, so that it registers itself with the ImR.

```
$ MessengerServer –ORBUseImR 1
Messenger server ready.
```

The ImR output shows that our server was automatically registered:

```
ImR: Server MessengerService is running at corbaloc:iiop:1.2@192.168.1.10:1323/
ImR: Auto adding NORMAL server:<MessengerService>
```

3. Run Client 1

```
$ MessengerClient messenger1.ior
message was sent
Reply was : A reply.
```

The ImR output shows that it forwarded the client to our server:

```
ImR: Forwarding invocation on <MessengerService> to <corbaloc:iiop:1.2@192.168.1.10:1323/…NUP…MessengerService…Messenger1> (Some parts of the IOR above were replaced with ellipsis for brevity.)
```

The server output shows a successful invocation:

```
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
```

4. Run Client 2

```
$ MessengerClient messenger2.ior
message was sent
Reply was : A reply.
```

The ImR output shows that it forwarded the client to our server:

```
ImR: Forwarding invocation on <MessengerService> to <corbaloc:iiop:1.2@192.168.1.10:8888/…NUP…MessengerService…Messenger2>
```
The server output is the same as above.

5. Create `corbaloc` object URLs

Here we demonstrate how to use the `tao_imr` utility to create an IOR, and also how to create one manually. We can create the IOR manually, because we started the ImR on a known endpoint.

```
$ tao_imr ior MessengerService/Messenger1 -f messenger3.ior
corbaloc:iiop:1.2@192.168.1.10:8888/MessengerService/Messenger1
$ echo corbaloc::localhost:8888/MessengerService/Messenger1 > messenger4.ior
```

6. Run Client 3

```
$ MessengerClient messenger3.ior
message was sent
Reply was : A reply.
```

The ImR output shows that this time we forward using a simple object key instead of a full-blown IOR. This means the server will have to forward the client again using its IOR Table.

```
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1272/MessengerService/Messenger1>
```

The server output is the same as above.

7. Run Client 4

```
$ MessengerClient messenger4.ior
message was sent
Reply was : A reply.
```

This time we can see that the ImR has to forward two invocations. This is because the server’s IOR Table contains an indirect object reference.

```
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1272/MessengerService/Messenger2>
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1272/\_NUP\_/MessengerService..Messenger2>
```

The server output is the same as above.
30.8 Advanced Examples

You may want to experiment with Activator example using the various Activation Modes, and other server start-up options. This should not require any source code changes. You can also experiment with the ImR persistence, start-up timeout, and ping interval settings.

You can find additional ImR tests and examples for TAO in
$TAO_ROOT/orbsvcs/examples/ImR and
$TAO_ROOT/orbsvcs/examples/ImpRepo.

30.9 Repository Persistence

The ImR can load and save its list of registered servers and Activators to persistent storage using one of three formats. The easiest to work with is an XML format that can be edited by hand. The XML file is rewritten in response to any change in registration information, and may therefore be inefficient for large, or very busy, repositories. A more efficient binary format is supported, but this can be difficult to work with, and is known to have problems on some platforms. The ImR can also save its registration information to the registry on Windows systems.

• XML

Starting the ImR with –x repo.xml creates a file containing an entry for every registered server and Activator. The schema for this XML file can be found at $TAO_ROOT/orbsvcs/ImplRepo_Service/ImR.xsd.

• Binary

Starting the ImR with –p repo.bin stores registered Activators and servers in a binary file.

Note

This option may have problems on some platforms when the memory mapped file used internally needs to be expanded. This problem may be fixed in future versions of TAO.

• Windows registry

Starting the ImR with –r stores registered servers and Activators at:
30.10 **ImR Utility**

TAO provides a command line tool called `tao_imr` that you can use to

- add or edit server information in the ImR,
- create IORs suitable for connecting to a server,
- view the status of registered servers,
- start or shutdown registered servers.

The full path to the `tao_imr` utility is `$ACE_ROOT/bin/tao_imr`.

30.10.1 **Command Line Options**

The general syntax for the ImR utility is `tao_imr command [options] [args].`

The commands and options that can be passed to `tao_imr` are summarized in the table below, followed by a detailed description of each command.

**Table 30-1 Command Line Options for `tao_imr`**

<table>
<thead>
<tr>
<th>Command</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activate</td>
<td></td>
<td>Start the server.</td>
</tr>
</tbody>
</table>
### 30.10.1.1 activate

**Synopsis**  
`tao_imr activate <server>`

This command is used to ensure that the specified server is activated, starting the server process, if necessary, using the Activator registered for the server. If the server is not already running, this requires that

<table>
<thead>
<tr>
<th>Command</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add / update</td>
<td>-a</td>
<td><strong>normal</strong> Start server on client invocation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>manual</strong> Start server with <code>tao_imr activate</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>per_client</strong> Start a new server for each client.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>auto_start</strong> Start server when ImR is started or with <code>tao_imr auto_start</code>.</td>
</tr>
<tr>
<td></td>
<td>-c cmdline</td>
<td>Command line string used to start server.</td>
</tr>
<tr>
<td></td>
<td>-e var=value</td>
<td>Set environment variable at server start-up. Repeat this option to add more than one environment variable.</td>
</tr>
<tr>
<td></td>
<td>-l activator</td>
<td>The activator to use for starting the server.</td>
</tr>
<tr>
<td></td>
<td>-w working_dir</td>
<td>Specify the server’s working directory.</td>
</tr>
<tr>
<td>ior</td>
<td>-f filename</td>
<td>Write a corbaloc URL to standard output. Optionally place the newly-created IOR into filename.</td>
</tr>
<tr>
<td>list</td>
<td>-v</td>
<td>Display the names and status of registered servers. Optionally display registered start-up information.</td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td>Remove the server registration.</td>
</tr>
<tr>
<td>shutdown</td>
<td></td>
<td>Shutdown the server.</td>
</tr>
<tr>
<td>shutdown-repo</td>
<td>-a</td>
<td>Shutdown the ImR, and optionally all registered activators.</td>
</tr>
</tbody>
</table>
1. The server has already been registered using add or update.
2. The registration information includes a command line and activator name.
3. A matching Activator is registered and running.

If any of the above conditions are not met, or if there is some problem activating the server, then an error message is returned.

**Note**  
*It is possible for the utility to report success when the ImR exceeds its ping retry count. In this case the server was not activated, and list –v will show that the server is locked.*

### 30.10.1.2 add/update

**Synopsis**

```
tao_imr add|update <server> [-a normal|manual|per_client|auto_start] [-c cmdline] [-e var=value] [-w working dir] [-r count] [-l activator]
```

These commands are used to add a new server registration and to update an existing server registration respectively. They take exactly the same options. The only difference is that `add` cannot be used to change an existing server. This means you can use `update` to add new servers.

You may specify an activation mode of `auto_start`, `manual`, `normal`, or `per_client`. The default activation mode is `normal`.

**Note**  
*It is possible for a client to make a second connection using the indirect object reference, and this will cause the ImR to launch another server. For this reason, per_client activation should be used with care.*

The command line, working directory, environment variables, and retry count are all optional, but you must specify at least a command line if you want the ImR to be able to start the server. The command line can be any command, and does not necessarily have to directly start the server. For example, it could run a script that causes the creation of a new persistent POA in an already running server process.

You can specify the `-e` option multiple times on a single command line to set multiple environment variables. The specified environment variables will replace any that currently exist when updating.
The working directory does not affect the command line. If the command is not in the path, then specifying the working directory does not allow the command to run. Therefore, you must also specify the directory to the cmdline argument.

The retry count is a feature to prevent further attempts to start servers that are not functioning correctly. If a server either cannot be started by the Activator, or does not register its running information with the ImR in a timely manor, then the ImR will attempt to start it again. The start-up timeout is a command line option to the ImR that is shared by all servers. An additional feature of the update command is that it always resets the start-up count for a server.

You can specify an activator for a server, and this defaults to the hostname of the machine on which the tao_imr utility is run. If you use the -n name feature of the Activator then you must use the same name here so that the ImR can find the correct Activator to start the server. Any auto-added servers will not have an Activator set, so the tao_imr update command must be used to set it if you want to make the server startable.

Note
In previous versions of the ImR, you could not change the Activator for a registered server. This restriction was due to an internal implementation detail that has since changed.

30.10.1.3 autostart

Synopsis
tao_imr autostart <server>

This command is used to start all servers registered with the auto_start activation mode. In other respects it works exactly the same as the activate command.

30.10.1.4 ior

Synopsis
tao_imr ior <server> [-f filename]

This command can be used to construct a valid simple corbaloc URL for accessing a server registered with the ImR. This is only useful if the address of the ImR is unknown, because, for example, you are using multicast to find the
Implementation Repository

ImR. If the address of the ImR is known, then it is easier to construct the URL manually using the form:

corbaloc:protocol:host_or_ip:port/ServerName

The ior command does not actually contact the ImR to lookup the address of the server. Instead it uses the first available protocol specified for the ImR connection.

30.10.1.5 list

Synopsis tao_imr list [-v]

Use this command to view the current status of all servers registered in the ImR. You will probably want to use the verbose option -v most of the time. All information registered using the add/update commands is displayed as well as the current running status of the server and whether the server is locked due to exceeding its retry count. If the server is locked, you can unlock it using the update command. For example:

tao_imr update myserver

30.10.1.6 remove

Synopsis tao_imr remove <server>

This command simply removes all information about the server from the ImR. If the server is running it is not shut down by this command. If the server is removed without being shut down first, then a NOT_FOUND exception will be caught by the server when it tries to unregister from the ImR. This exception is ignored, but an error message is displayed.

30.10.1.7 shutdown

Synopsis tao_imr shutdown <server>

This command shuts down a running server by using the ServerObject that every server internally registers with the ImR at start-up. The orb->shutdown(0) operation is called in the server, typically causing the
server to return from `orb->run()` and shut down gracefully. Note that this means servers with multiple persistent POAs can be shut down using any one of the POA names. If this behavior is not desired, then you should use separate ORBs in the server.

### 30.10.1.8 shutdown-repo

**Synopsis**

```
tao_imr shutdown-repo [-activators]
```

This command shuts down the ImR cleanly. This can also be done by using Ctrl-C, or sending a `SIGINT` or `SIGTERM` to the ImR and/or Activator processes.

The `-a` option specifies that you also want to attempt to shut down any registered Activators. If any Activator cannot be shut down, it is ignored, and no error is returned.

### 30.10.2 Examples

For simplicity, we assume the `ImplRepoServiceIOR` environment variable has been set to the appropriate IOR. This environment variable allows applications, such as our server and the `tao_imr` utility, to find the ImR without the need to supply lengthy `-ORBInitRef` options on the command line. Alternatively, we could start the ImR with `-m` to allow clients to find it using multicast service discovery.

#### 30.10.2.1 list existing servers

```
$ tao_imr list
No servers found.
```

#### 30.10.2.2 add a new server

```
$ tao_imr add mysrv -l myhost -c mysrv -w $SERVERS/mysrv -e env1=1 -e env2=2 -a per_client -r 1
Successfully registered server <mysrv>
```

#### 30.10.2.3 list again

```
$ tao_imr list
<mysrv>
$tao_imr list -v
Server <mysrv>
```
Activator: myhost
Command Line: mysrv
Working Directory: $SERVERS/mysrv
Activation Mode: PER_CLIENT
Number of retries: 1
Environment Variable: env1=1
Environment Variable: env2=2
No running info available for PER_CLIENT mode

30.10.2.4 update the server
$ tao_imr update -l newhost -a normal
Can not change the Activator for <mysrv>.

$ tao_imr update -a normal
Successfully updated <mysrv>.

30.10.2.5 start the server
$ tao_imr activate mysrv
Cannot activate server <TestMessengerService>, reason: <Cannot start server.>

$ tao_imr update mysrv -c "mysrv -ORBUseIMR 1"
Successfully updated <mysrv>

$ tao_imr activate mysrv
Successfully activated server <mysrv>

30.10.2.6 shutdown the server
$ tao_imr shutdown mysrv
Successfully shut down server <mysrv>

30.10.2.7 create an ior
$ tao_imr ior mysrv

30.10.2.8 remove the server
$ tao_imr remove mysrv
Successfully removed server <mysrv>
30.11 ImplRepo_Service

ImplRepo_Service is the primary application that we generally refer to as the ImR. It is responsible for maintaining a repository of server and activator information, and using it to support indirect binding of CORBA objects.

If the ImR is not started with multicast service discovery support, then you must provide some mechanism for other processes to find the ImR when they use `resolve_initial_references()`. This requires the servers, Activators, and tao_imr to be started with an `-ORBInitRef` option, such as:

```
-ORBInitRef ImplRepoService=corbaloc::host:port/ImR
-ORBInitRef ImplRepoService=corbaloc::host:port/ImplRepoService
-ORBInitRef ImplRepoService=file://ImR.ior
-ORBDefaultInitRef corbaloc::host:port
```

Alternatively, you can set the `ImplRepoServiceIOR` environment variable:

```
$ export ImplRepoServiceIOR=corbaloc::host:port/ImR
```

For most of the above to work, the ImR must be started at a known endpoint.

```
-ORBListenEndpoints iiop://:8888
-ORBListenEndpoints iiop://host:port
```

The full path to the ImR is:

```
$TAO_ROOT/orbsvcs/ImplRepo_Service/Impl_Repo_Service
```

30.11.1 Command Line Options

**Synopsis**


<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h</code> or <code>-?</code></td>
<td>Display help/usage.</td>
</tr>
<tr>
<td><code>-d level</code></td>
<td>Specify IMR specific debugging level, 0-2. Default is 1.</td>
</tr>
</tbody>
</table>
There are currently three valid settings for the debug level.

- **0**
  - most output is disabled
- **1**
  - very basic information is displayed, with usually just one output per interesting operation
- **2**
  - more messages are displayed, and more details are displayed for existing messages

In practice, debug level 1 is probably the best choice for day-to-day usage, while level 2 is most useful when trying to resolve problems.

The `-l` option prevents the ImR repository information from being modified by the `tao_imr` `add` or `update` commands. However, server
auto-registration, Activator registration, and server running status are all still persisted to the repository.

The `-m` option provides a convenient way for all servers and Activators to find the ImR without the need to pass IORs, use `corbaloc` object URLs, or use the `-ORBInitRef` option. Instead, a multicast endpoint is established when the ImR starts, and servers and Activators automatically find the correct IOR using multicast. The multicast port may be specified using:

```
-ORBMulticastDiscoveryEndpoint <port>
```

If this option is not specified, then the environment variable `ImplRepoServicePort` is used. The default port number is `10018`.

The output file for the `-o` option is a simple stringified IOR stored in a text file, suitable for use as a `file://` object URL.

The `-v` option allows the ImR to more efficiently work with servers. If the ImR has successfully pinged a server within the specified number of milliseconds, then it assumes that the server is still running. Setting the ping interval to zero disables the ping feature completely, and servers are assumed to be running if they register a running status with the ImR.

---

**Note:** The current implementation for server registration causes servers to notify the ImR of their running status upon POA creation. This allows the ImR to forward clients to servers that are not yet ready for use, and clients may get TRANSIENT exceptions indicating that the POA is in a holding state. If your clients cannot handle this, then you should probably avoid disabling the ping feature. This problem may be fixed in future versions of TAO.

---

The `-t` option allows the user to set a timeout value, in seconds, by which the ImR will wait before giving up on starting a server. If the ImR does not receive running information from a server within the specified timeout interval, it will abort the server start-up. Depending on the server’s retry count setting, the ImR may attempt to launch the server again. Once the retry count has been exceeded, the ImR will not attempt to start the server again until an administrator uses `tao_imr update` to reset the server’s start-up counter. The start-up timeout value can be disabled by specifying `-t 0` when starting the ImR. However, disabling the start-up timeout is not recommended as the ImR...
could become blocked permanently waiting for a server to start. The default start-up timeout value is 60 seconds.

In addition to the normal start-up options, the \texttt{--install} or \texttt{--remove} options can be used to install or remove the ImR as a Windows service. When installing, all other command line options are saved in the Windows registry under:

\texttt{HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\TAOImR}

These options are used when the service is started.

### 30.11.2 Examples

Start the ImR with default options:

\>
ImplRepo\_Service.exe
Implementation Repository: Running
  
  Ping Interval : 10000ms
  Startup Timeout : 60s
  Persistence : Disabled
  Multicast : Disabled
  Debug : 1
  Locked : False

Start with alternative options:

\>
ImplRepo\_Service.exe -v 500 -t 5 -x repo.xml -m -d 2 -l
Implementation Repository: Running
  
  Ping Interval : 500ms
  Startup Timeout : 5s
  Persistence : repo.xml
  Multicast : Enabled
  Debug : 2
  Locked : True

Install as a Windows service with several non-default options

\>
ImplRepo\_Service -v 1000 -t 30 -p repo.bin -ORBListenEndpoints iiop://:8888 -c install

Start a server

\>
myserver -ORBUseIMR 1 -ORBInitRef ImplRepoService=file://imr.ior
30.12 ImR_Activator

The Activator is an extremely simple process-starting service. It accepts start-up information from the ImR, and attempts to launch a process.

**Note**  
On UNIX and UNIX-like platforms the Activator can detect when spawned processes terminate, and can optionally notify the ImR when this happens.

Once started, the server registers itself with the ImR directly. Each persistent POA within the server registers itself as a separate server within the ImR.

The start-up information passed to the Activator does not necessarily have to directly start a server. For example, it could run a script that causes the creation of a new persistent POA in an already running server process.

At start-up, the ImR Activator tries to register itself with an ImR. If an ImR cannot be found, the Activator will not be able to notify the ImR when it is shut down or when spawned processes are terminated.

The full path to the ImR_Activator is $TAO_ROOT/orbsvcs/ImplRepo_Service/ImR_Activator.

### 30.12.1 Command Line Options

**Synopsis**  
ImR_Activator [-h|-?] [-d 0|1|2] [-o filename] [-n name] [-c command]

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or -?</td>
<td>Display help/usage.</td>
</tr>
<tr>
<td>-d level</td>
<td>Specify IMR specific debugging level, 0-2. Default is 1.</td>
</tr>
<tr>
<td>-o filename</td>
<td>Write the Activator IOR to filename.</td>
</tr>
<tr>
<td>-n name</td>
<td>The name of the Activator. Default is the name of the host on which the Activator is running.</td>
</tr>
</tbody>
</table>
There are currently three valid settings for the debug level:

- **0**
  
  Most output is disabled.

- **1**
  
  Very basic information is displayed, with usually just one output per interesting operation.

- **2**
  
  More messages are displayed, and more details are displayed for existing messages.

In practice, debug level 1 is probably the best choice for day-to-day usage, while level 2 is most useful when trying to resolve problems.

The output file for the \(-o\) option is a simple stringified IOR stored in a text file, suitable for use as a file:// object URL. This option is not very useful, because typically the ImR is the only client for the Activator.

You can change the name of an Activator instance, which allows running more than one Activator on a single machine. This probably is not of much practical value unless you were to create your own Activator class that behaves differently from the default simple process launching service. For example, you might create an activator that spawns servers as threads within its own process.

In addition to the normal start-up options, the \(-c\) install, \(-c\) remove, and \(-c\) install_no_imr options can be used to install or remove the Activator as an Windows service. When installing, all other command line options are saved in the Windows registry under:

\[ HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TAOImR \]
These options are used when the service is started. The `install_no_imr` option should be used when installing the Activator on a separate machine with no ImR, otherwise the installation will ensure that the ImR is always started before the Activator by setting a dependency between the two services.

**Note**

*Only a single Activator can be installed as a Windows service due to the way the start-up options are stored in the registry.*

### 30.12.2 Examples

Start the ImR Activator with default options:

```
>ImR_Activator.exe
ImR Activator: Starting MYHOST
... Multicast error message omitted
ImR Activator: Not registered with ImR.
```

Start the ImR Activator with debug level 2, store the Activator’s stringified IOR in a file, and register with an ImR that was started with multicast discovery enabled:

```
>ImR_Activator.exe -d 2 -o activator.ior
ImR Activator: Starting MYHOST
ImR Activator: Registered with ImR.
```

Install the ImR Activator as a Windows service on a machine with no ImR:

```
>ImR_Activator.exe -d 2 -o activator.ior -c install_no_imr
```
CHAPTER 31

Data Distribution Service

31.1 Introduction

The OMG Data Distribution Service for Real-Time Systems specification (OMG Document formal/04-12-02) defines a service for efficiently distributing application data between participants in a distributed application. This service is not specific to CORBA. The specification provides a platform independent model (PIM) as well as a platform specific model (PSM) that maps the PIM onto a CORBA IDL implementation. The service is divided into two components: the Data-Centric Publish-Subscribe (DCPS) layer and an optional Data Local Reconstruction Layer (DLRL). The DCPS layer transports data from publishers to subscribers according to Quality of Service constraints associated with the data topic, publisher, and subscriber. The DLRL allows distributed data to be shared by local objects located remotely from each other as if the data were local. The DLRL is built on top of the DCPS layer.

Note  
TAO currently implements a subset of the DCPS layer. None of the DLRL functionality currently exists in the TAO DDS implementation.
The DCPS layer provides another publish-subscribe API for applications that is conceptually similar to the OMG Event and Notification Services as well as the TAO Real-Time Event Service. The main difference with DCPS is that it only specifies CORBA IDL interfaces for the set up, control, and configuration of the application and assumes that the data transmission occurs via mechanisms other than CORBA. This enables DDS implementations to achieve higher performance and better quality of service than the CORBA-based alternatives mentioned above.

This chapter focuses on TAO’s DDS implementation and the features that it implements. For additional details about DDS, developers should refer to the DDS specification (OMG Document formal/04-12-02) as it contains in-depth coverage of all the service’s features, including those not supported by TAO’s current DDS implementation.

31.2 DCPS Overview

In this section we introduce the main concepts and entities of the DCPS layer and discuss how they interact and work together.

31.2.1 Basic Concepts

Figure 31-1 shows an overview of the DDS DCPS layer. The following subsections define the concepts shown in this diagram.
Figure 31-1 DCPS Conceptual Overview
31.2.1.1 Domain
The domain is the fundamental partitioning unit within DCPS. Each of the other entities belongs to a domain and can only interact with other entities in that same domain. Application code is free to interact with multiple domains but must do so via separate entities that belong to the different domains. Each process interacting within a domain is referred to as a domain participant.

31.2.1.2 Topic
The topic is the fundamental means of interaction between publishing and subscribing applications. Each topic has a unique name within the domain and a specific data type that it publishes. Each topic data type can specify zero or more fields that make up its key. When publishing data, the publishing process always specify the topic. Subscribers request data via the topic. In DCPS terminology you publish individual data samples for different instances on a topic. Each instance is associated with a unique value for the key. A publishing process publishes multiple data samples on the same instance by using the same key value for each sample.

31.2.1.3 Data Writer
The data writer is used by the publishing application code to pass values to the DDS. Each data writer is bound to a particular topic. The application uses the data writer’s type-specific interface to publish samples on that topic. The data writer is responsible for marshaling the data and passing it to the publisher for transmission.

31.2.1.4 Publisher
The publisher is responsible for taking the published data and disseminating it to all relevant subscribers in the domain. The exact mechanism employed is left to the service implementation.

31.2.1.5 Subscriber
The subscriber receives the data from the publisher and passes it to any relevant data readers that are connected to it.
31.2.1.6 Data Reader
The data reader takes data from the subscriber, demarshals it into the appropriate type for that topic, and delivers the sample to the application. Each data reader is bound to a particular topic. The application uses the data reader’s type-specific interfaces to receive the samples.

31.2.2 Built-In Topics
The DDS specification defines a number of topics that are built-in to the DDS implementation. Subscribing to these built-in topics gives application developers access to the state of the domain being used including which topics are registered, which publishers and subscribers are connected and disconnected, and the QoS settings of the various entities. While subscribed, the application receives samples indicating changes in the entities within the domain.

The following table shows the built-in topics defined within the DDS specification:

<table>
<thead>
<tr>
<th>Topic Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPSParticipant</td>
<td>Each instance represents the a domain participant.</td>
</tr>
<tr>
<td>DCPSTopic</td>
<td>Each topic is an instance.</td>
</tr>
<tr>
<td>DCPSPublication</td>
<td>Each instance represents a data writer</td>
</tr>
<tr>
<td>DCPSSubscription</td>
<td>Each instance represents a data reader.</td>
</tr>
</tbody>
</table>

Figure 31-2 Built-In Topics

31.2.3 Quality of Service Policies
The DDS specification defines a number of Quality of Service (QoS) policies that are used by applications to specify their QoS requirements to the service. Participants specify what behavior they require from the service and the service decides how to achieve these behaviors. These policies can be applied to the various DCPS entities (Topic, Data Writer, Data Reader, Publisher, Subscriber, Domain Participant) although not all policies are valid for all types of entities.

Subscribers and publishers collaborate to specify QoS through an offer-request paradigm. Publishers offer a set of QoS policies to all subscribers. Subscribers request a set of policies that they require. The DDS implementation then attempts to match the requested policies with the offered
policies. If the policies are consistent the subscription is initiated. If the policies are not consistent then the subscription attempt fails.

The QoS policies currently implemented by TAO are discussed in detail in 31.6.

### 31.2.4 Listeners

The DPCS layer defines a callback interface for each entity that allows an application processes to “listen” for certain state changes or events pertaining to that entity. For example, a Data Reader Listener is notified when there are data values available for reading.

### 31.2.5 Conditions

**Note**  
*TAO DDS does not currently support conditions.*

Conditions and wait-sets allow an alternative to listeners in detecting events of interest in DDS. The general pattern is

- The application creates a specific kind of condition object, such as a Read Condition, and attaches it to a Wait Set.
- The application waits on the Wait Set until one or more Conditions become true.
- The application calls operations on the corresponding entity objects to extract the necessary information.

### 31.3 TAO DDS Implementation

### 31.3.1 Compliance

Appendix A of the DDS specification defines five compliance points for a DDS implementation:

1. Minimum Profile
2. Content-Subscription Profile
3. Persistence Profile
4. Ownership Profile
5. Object Model Profile

TAO’s current DDS implementation is working towards the Minimum Compliance Profile.

### 31.3.2 TAO DDS Architecture

This section gives a brief overview of the TAO DDS implementation, some of its features, and some of its components. The `$DDS_ROOT` environment variable should point to the base directory of the TAO DDS distribution, typically the `$ACE_ROOT/DDS` directory. Source code for TAO’s DDS implementation can be found under `$DDS_ROOT/dds`. DDS tests can be found under `$DDS_ROOT/tests`.

#### 31.3.2.1 Pluggable Transport Layer

TAO’s DDS implementation uses the CORBA interfaces defined by the DDS specification to initialize and control service usage. Data transmission is accomplished via a TAO-specific *Pluggable Transport* layer that allows the service to be used with a variety of transport protocols. TAO DDS currently implements simple TCP and UDP transports. Transports are created via a factory object and are associated with publishers and subscribers who use them for their data transmission.

The pluggable transport layer enables application developers to implement their own customized protocols. Implementing your own custom transport involves specializing a number of classes defined in the transport framework directory `$DDS_ROOT/dds/DCPS/transport/framework`. See the simple TCP implementation in `$DDS_ROOT/dds/DCPS/transport/SimpleTCP` for details.
31.3.2.2 Custom Marshaling

Because data transmission is not done with CORBA, DDS implementations are free to marshal the data using customized formats. TAO’s implementation uses a more efficient variation of CORBA’s Common Data Representation (CDR). A new IDL compiler switch (-Gdcps) causes the TAO IDL compiler to generate the appropriate marshaling and instance key support code for DCPS-enabled types.

31.3.2.3 DCPS Information Repository

The DCPS Information Repository is a server that acts as the intermediary or broker between the publisher and subscriber. When a client requests a
subscription for a topic, the DCPS Information Repository locates the topic and notifies any existing publishers of the location of the new subscriber. This process needs to be running whenever TAO DDS is being used.

### 31.3.2.4 Threading
TAO DDS creates its own ORB as well as a separate thread upon which to run that ORB. It also uses its own threads to process incoming and outgoing non-CORBA transport I/O. Your application may get called back from these threads via the Listener mechanism of DCPS.

When publishing a sample via DDS, TAO DDS attempts to send the sample to any connected subscribers using the calling thread. If the send call blocks, then the sample may be queued for sending on a separate service thread. This behavior depends on the QoS policies described in 31.6.

All incoming data in the subscriber is read by the service thread and queued for reading by the application. Data reader listeners are called from the service thread.

### 31.4 Using DCPS

This section focuses on an example application using DCPS to distribute data from a publisher process to a subscriber. It is based on a simple messenger application where a single publisher publishes messages and a single subscriber subscribes to them. We use the default QoS properties and the Simple TCP transport. Full source code for this example is in the TAO 1.4a source code distribution in the directory $DDS_ROOT/DevGuideExamples/DDS/Messenger. Additional DDS and DCPS features are discussed in later sections.

#### 31.4.1 Defining the Data Types

Each data type used by DDS is defined using IDL. TAO’s DDS implementation uses `#pragma` statements to identify the data types that DDS processes. These data types are processed by the TAO IDL compiler and the `dcps_ts.pl` script to generate code necessary for transmitting these types with DDS. Here is the IDL file that defines our `Message` data type:

```idl
module Messenger {
```
#pragma DCPS_DATA_TYPE "Messenger::Message"
#pragma DCPS_DATA_KEY "Messenger::Message subject_id"

struct Message {
  string from;
  string subject;
  long subject_id;
  string text;
  long count;
};

The DCPS_DATA_TYPE pragma marks a data type for use with DDS. A fully scoped type name must be used with this pragma. Currently, TAO’s implementation requires the data type to be a structure. The structure may contain scalar types (short, long, float, etc.), enumerations, strings, sequences, arrays, structures, and unions. This example defines the structure Message in the Messenger module for use in this DDS example.

The DCPS_DATA_KEY pragma identifies a field of the DDS data type that is used as the key for this type. A data type may have zero or more keys. These keys are used to identify the different instances within a topic that use this type. The key should be a numeric or enumerated type. The pragma is passed the fully scoped type name and the member name that is the key for that type. In the above example, we identify the subject_id member of Messenger::Message as the key. Each message published with a unique subject ID value is defined as a different instance within a topic. Subsequent samples with the same subject ID value are treated as replacement values for that instance.

31.4.2 Processing the IDL

The DDS IDL is processed like any other IDL with the exception that we pass the -Gdcps option the TAO IDL compiler.

tao_idl -Gdcps Messenger.idl

This causes the IDL compiler to generate additional serialization and key support code that DDS uses to marshal and demarshal the Message structure. In addition, we need to process the IDL file with the dcps_ts.pl script to generate the required type support code for the data readers and writers. This
script is located in $DDS_ROOT/bin and generates three files for each 
DCPS_DATA_TYPE pragma encountered. The three files all begin with the data 
type name (without enclosing scopes) and are appended as follows:

- `<type>Typesupport.idl`
- `<type>TypesupportImpl.h`
- `<type>TypesupportImpl.cpp`

For example, running `dcps_ts.pl` as follows:

`dcps_ts.pl Messenger.idl`

generates `MessageTypeSupport.idl`, `MessageTypeSupportImpl.h`, and 
`MessageTypeSupportImpl.cpp`. The IDL file contains the 
`MessageTypeSupport`, `MessageDataWriter`, and `MessageDataReader` 
interface definitions. These are type-specific DDS interfaces that we use later 
to register our data type with the domain, publish samples of that data type, 
and receive published samples. The implementation files contain servant 
implementations for these interfaces. The generated IDL file should itself be 
compiled to generate stubs and skeletons. These and the implementation file 
should be linked with your DDS applications that use the `Message` type. This 
type support generation script has a number of options that specialize the 
generated code. These options are described in 31.10.

Typically, you do not directly invoke the IDL compiler or `dcps_ts.pl` script 
as above, but let your build environment do it for you. The entire process is 
simplified when using MPC, by inheriting from the `dcps` project. Here is the 
MPC file used by this example’s publisher:

```mpc
project('Publisher') : taoexe, portablesrv, dcps {
    requires += exceptions
    exename = publisher

    TypeSupport_Files {
        Messenger.idl >> MessageTypeSupport.idl MessageTypeSupportImpl.h 
        MessageTypeSupportImpl.cpp
    }

    IDL_Files {
        Messenger.idl
        MessageTypeSupport.idl
    }

    Source_Files {
```
The dcps parent project adds the -Gdcps IDL compiler option, links in the DCPS library, and adds the Type Support custom build rules. The TypeSupport_Files section above tells MPC to generate the Message type support files from Messenger.idl using the dcps_ts.pl script.

### 31.4.3 Starting the DCPS Information Repository

The DCPS Information Repository server is found in $DDS_ROOT/dds/InfoRepo. This process hosts the DCPSInfo CORBA object that is the entry point for all DDS functionality. The mechanism for locating the DCPSInfo object is the IOR written out when the server is started. We can alter the file name used with the -o option.

```
$DDS_ROOT/dds/InfoRepo/DCPSInfoRepo -o repo.ior
```

The full set of command line options for this server are documented in 31.11.

### 31.4.4 A Simple Message Publisher

In this section we will step through the setup of a simple DDS publication process. The code is broken into logical sections and explained as we present each section. We omit some uninteresting sections of the code (such as `#include` directives, error handling, and cross-process synchronization). To see the full code see the publisher.cpp and Writer.cpp files in $DDS_ROOT/DevGuideExamples/DDS/Messenger.

The first section of `main()` initializes this process as a DDS participant.

```c++
int main (int argc, char *argv[]) {
    try {
        DDS::DomainParticipantFactory_var dpf = TheParticipantFactoryWithArgs(argc, argv);
        DDS::DomainParticipant_var participant = dpf->create_participant(411, // domain ID
            PARTICIPANT_QOS_DEFAULT,
            DDS::DomainParticipantListener::_nil());
        if (CORBA::is_nil (participant.in())) {
            std::cerr << "create_participant failed." << std::endl;
        } else {
            // Initialize the participant...
        }
    } catch (...) {
        // Handle exceptions...
    }
}
```
The ParticipantFactoryWithArgs macro is defined in Service_Participant.h and initializes the Domain Participant Factory with the command line arguments. These command line arguments are used to initialize the ORB that the DDS service uses as well as the service itself. This allows us to pass ORB_init() options on the command line as well as DDS configuration options of the form -DCPS*. These options are fully described in 31.9. The create_participant() operation uses the domain participant factory to register this process as a participant in the domain specified by the ID of 411. The participant uses the default QoS policies and no listeners.

The Domain Participant object reference returned is then used to register our Message data type.

```cpp
MessageTypeSupportImpl* servant = new MessageTypeSupportImpl();
if (DDS::RETCODE_OK != servant->register_type(participant.in (),
    "Message")) {
    std::cerr << "register_type failed." << std::endl;
    return 1;
}
```

First, we create a MessageTypeSupportImpl object, then register it with the domain using the name “Message”. Next, we create the topic using the participant.

```cpp
DDS::TopicQos topic_qos;
participant->get_default_topic_qos(topic_qos);
DDS::Topic_var topic =
    participant->create_topic ("Movie Discussion List",
    "Message",
    topic_qos,
    DDS::TopicListener::_nil());
if (CORBA::is_nil(topic.in())) {
    std::cerr << "create_topic failed." << std::endl;
    return 1;
}
```

This creates a topic named “Movie Discussion List” with a type of “Message” and the default QoS policies.

We may now initialize the Simple TCP transport we want to use.

```cpp
TheTransportFactory->register_type(TCP_TYPE_ID,
```
new TAO::DCPS::SimpleTcpFactory();

TAO::DCPS::SimpleTcpConfiguration_rch writer_config =
new TAO::DCPS::SimpleTcpConfiguration();
TAO::DCPS::TransportImpl_rch tcp_impl =
TheTransportFactory->create(TCP_IMPL_ID, TCP_TYPE_ID);
if (tcp_impl->configure(writer_config.in()) != 0) {
    std::cerr << "Failed to configure the transport." << std::endl;
    return 1;
}

This code registers the Simple TCP Factory with the transport factory singleton using the TCP_TYPE_ID. Then we can use this type ID to create a transport implementation. Finally, we configure the transport implementation with the default configuration object. Now we are ready to create the publisher and attach the transport implementation we want it to use.

DDS::Publisher_var pub =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT,
    DDS::PublisherListener::_nil());
if (CORBA::is_nil(pub.in())) {
    std::cerr << "create_publisher failed." << std::endl;
    return 1;
}

// Attach the publisher to the transport.
TAO::DCPS::PublisherImpl* pub_impl =
    reference_to_servant< TAO::DCPS::PublisherImpl, DDS::Publisher_ptr>(pub);
if (0 == pub_impl) {
    std::cerr << "Failed to obtain publisher servant" << std::endl;
    return 1;
}

TAO::DCPS::AttachStatus status = pub_impl->attach_transport(tcp_impl.in());

We need to do the attach_transport() on the servant and not on the CORBA object. The CORBA object implements the OMG-defined Publisher interface and lacks any TAO-specific pluggable transport functionality. The reference_to_servant function template is a convenience function for navigating from the CORBA object reference of an entity to the servant. This works because the DDS service collocates the publisher object in the publication process.

With the publisher in place, we create the data writer.

    // Create the datawriter
    DDS::DataWriterQos dw_qos;
pub->get_default_datawriter_qos (dw_qos);

DDS::DataWriter_var dw =
  pub->create_datawriter(topic.in (),
  dw_qos,
  DDS::DataWriterListener::_nil());
if (CORBA::is_nil(dw.in())) {
  std::cerr << "create_datawriter failed." << std::endl;
  return 1;
}

When we create the data writer we pass the topic object reference, the default QoS policies, and a null listener reference. Now we can register the instance we wish to publish. We’ll narrow the data writer reference to a MessageDataWriter object reference so we can use the type-specific registration and publication operations.

MessageDataWriter_var message_dw = MessageDataWriter::_narrow(writer.in());

Messenger::Message message;
message.subject_id = 99;
DDS::InstanceHandle_t handle = message_dw->_cxx_register (message);

After we populate the Message structure we called the _cxx_register() function to register the instance. The instance is identified by the subject_id value of 99 (because we earlier specified that field as the key). We later use the returned instance handle when we publish a sample.

Note This registration operation is actually register() in IDL but because register is a C++ keyword, the CORBA C++ mapping maps the operation to the _cxx_register() member function.

The example code waits for the subscriber to become connected and fully initialized. Once this is completed, the message publication is quite straightforward:

//Populate instance
message.from       = CORBA::string_dup("Comic Book Guy");
message.subject    = CORBA::string_dup("Review");
message.text       = CORBA::string_dup("Worst. Movie. Ever.");
message.count      = 0;
DDS::ReturnCode_t ret = message_dw->write(message, handle);
This message is distributed to all connected subscribers that are registered for our topic. The second argument to `write()` specifies the instance we are publishing the sample upon. It should be passed either a handle returned by `_cxx_register()` or `DDS::HANDLE_NIL`. Passing a `DDS::HANDLE_NIL` value indicates that the data writer should determine the instance by inspecting the key of the sample.

### 31.4.5 Setting up the Subscriber

A lot of the subscriber’s code is identical or analogous to the publisher that we just finished exploring. We will progress quickly through the similar parts and refer you to the discussion above for details. The beginning of the subscriber is identical to the publisher as we initialize the service and join our domain.

```cpp
int main (int argc, char *argv[]) {
    try {
        DDS::DomainParticipantFactory_var dpf =
            TheParticipantFactoryWithArgs(argc, argv);
        DDS::DomainParticipant_var participant =
            dpf->create_participant(411, // Domain ID
                PARTICIPANT_QOS_DEFAULT,
                DDS::DomainParticipantListener::_nil());
        if (CORBA::is_nil (participant.in ())) {
            std::cerr << "create_participant failed." << std::endl;
            return 1;
        }
        
        Then the message type and topic are initialized. Note that if the topic has already been initialized in this domain with the same data type and compatible QoS, the `create_topic()` invocation returns a reference corresponding to the existing topic. If the type or QoS specified in our `create_topic()` invocation do not match that of the existing topic then the invocation fails. There is also a `find_topic()` operation our subscriber could use to simply retrieve an existing topic.

        MessageTypeSupportImpl* mts_servant = new MessageTypeSupportImpl();
        if (DDS::RETCODE_OK != mts_servant->register_type(participant.in (),
            "Message") ( std::cerr << "Failed to register the MessageTypeSupport." << std::endl;
            return 1;
        )
    }
    
    DDS::TopicQos topic_qos;
```
participant->get_default_topic_qos(topic_qos);
DDS::Topic_var topic =
    participant->create_topic("Movie Discussion List",
    "Message",
    topic_qos,
    DDS::TopicListener::_nil());
if (CORBA::is_nil(topic.in())) {
    std::cerr << "Failed to create_topic." << std::endl;
    return 1;
}

Now we initialize the Simple TCP transport the same way as in the publisher.

// Initialize the transport
TheTransportFactory->register_type(TCP_TYPE_ID,
    new TAO::DCPS::SimpleTcpFactory());
TAO::DCPS::SimpleTcpConfiguration_rch reader_config =
    new TAO::DCPS::SimpleTcpConfiguration();
TAO::DCPS::TransportImpl_rch tcp_impl =
    TheTransportFactory->create(TCP_IMPL_ID, TCP_TYPE_ID);
if (tcp_impl->configure(reader_config.in()) != 0) {
    std::cerr << "Failed to init_reader_transport." << std::endl;
    return 1;
}

Next we can create the subscriber with the default QoS and then attach the
Simple TCP object to the subscriber servant.

// Create the subscriber and attach to the corresponding
// transport.
DDS::Subscriber_var sub =
    participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT,
    DDS::SubscriberListener::_nil());
if (CORBA::is_nil(sub.in())) {
    std::cerr << "Failed to create_subscriber." << std::endl;
    return 1;
}

// Attach the subscriber to the transport.
TAO::DCPS::SubscriberImpl* sub_impl =
    reference_to_servant< TAO::DCPS::SubscriberImpl,
    DDS::Subscriber_ptr> (sub.in());
if (0 == sub_impl) {
    std::cerr << "Failed to obtain subscriber servant\n" << std::endl;
    return 1;
}
TAO::DCPS::AttachStatus status = sub_impl->attach_transport(tcp_impl.in());
We need to attach a listener object to the data reader we create, so we can use it to detect when data is available. The code below constructs the listener servant and activates a listener CORBA object. The `DataReaderListenerImpl` class is shown in the next subsection.

```cpp
// activate the listener
DataReaderListenerImpl listener_servant;
PortableServer::POA_var poa = TheServiceParticipant->the_poa();
CORBA::Object_var obj = poa->servant_to_reference(&listener_servant);
DDS::DataReaderListener_var listener =
    DDS::DataReaderListener::_narrow(obj.in());
if (CORBA::is_nil(listener.in())) {
    std::cerr << "listener is nil." << std::endl;
    return 1;
}
```

Now we can create the data reader, associating it with our topic, the default QoS properties, and the listener object we just created.

```cpp
// Create the Datareaders
DDS::DataReaderQos dr_qos;
sub->get_default_datareader_qos(dr_qos);
DDS::DataReader_var dr = sub->create_datareader(topic.in(),
    dr_qos,
    listener.in());
if (CORBA::is_nil(dr.in())) {
    std::cerr << "create_datareader failed." << std::endl;
    return 1;
}
```

Now this thread is free to do any application work we want while our listener object should get called from one of the service’s threads when a sample is available.

### 31.4.6 The Data Reader Listener Servant

Our listener servant implements the `DataReaderListener` interface from `$DDS_ROOT/dds/DdsDcpsSubscription.idl`. This interface defines a number of operations we must implement each of which is invoked to inform us of different events. Here is the interface definition:

```cpp
interface DataReaderListener : Listener {
    void on_requested_deadline_missed(in DataReader reader,
        in RequestedDeadlineMissedStatus status);
    void on_requested_incompatible_qos(in DataReader reader,
```
This servant class is defined like any other CORBA servant by inheriting from
POA_DDS::DataReaderListener and implementing all the virtual
functions. We stub out most of these listener operations with simple print
statements. The only operation truly needed for this example is
on_data_available() and it is the only function of this servant we need to
explore.

```cpp
void DataReaderListenerImpl::on_data_available(DDS::DataReader_ptr reader)
throw (CORBA::SystemException)
{
    num_reads_ ++;
    try {
        MessageDataReader_var message_dr = MessageDataReader::_narrow(reader);
        if (CORBA::is_nil(message_dr.in())) {
            std::cerr << "read: _narrow failed." << std::endl;
            return;
        }
        Messenger::Message message;
        DDS::SampleInfo si;
        DDS::ReturnCode_t status = message_dr->take_next_sample(message, si) ;
        if (status == DDS::RETCODE_OK) {
            std::cout << "Message: subject    = " << message.subject.in() << std::endl
                       << "         subject_id = " << message.subject_id   << std::endl
                       << "         from       = " << message.from.in()    << std::endl
                       << "         count      = " << message.count        << std::endl
                       << "         text       = " << message.text.in()    << std::endl;
            std::cout << "SampleInfo.sample_rank = " << si.sample_rank << std::endl;
        } else if (status == DDS::RETCODE_NO_DATA) {
```

std::cerr << "ERROR: reader received DDS::RETCODE_NO_DATA!" << std::endl;
} else {
    std::cerr << "ERROR: read Message: Error: " <<  status << std::endl;
}
} catch (CORBA::Exception& e) {
    std::cerr << "Exception caught in read:" << std::endl << e << std::endl;
}

If additional samples are available, the service calls this function again. Reading values a single sample at a time is not the most efficient way to process incoming data. The Data Reader interface gives a number of different options for processing data in a more efficient manner. We’ll look at some of these operations in 31.5.

### 31.4.7 Cleaning up in DDS Clients

After we are finished in the publisher and subscriber, we can use the following code to clean up the DDS-related objects:

```c++
participant->delete_contained_entities();
dpf->delete_participant(participant.in ());
TheTransportFactory->release();
TheServiceParticipant->shutdown ();
```

The domain participant’s `delete_contained_entities()` operation deletes all the topics, subscribers, and publishers created with that participant. Once this is done, we can use the domain participant factory to delete our domain participant. Lastly, we release our transport factory and shutdown the service participant.

### 31.4.8 Running the Example

This example can be run with the following commands.

```
$DDS_ROOT/dds/InfoRepo/DCPSInfoRepo -NOBITS -o repo.ior -d domain_ids
./publisher
./subscriber
```

Running each of these commands in its own window should enable you to most easily understand the output. One side effect of using the default QoS properties is that as we increase the number of samples being published some of the samples will be dropped as the subscriber falls behind. If we don’t want
this to happen we need to either ensure that the subscriber can keep up or change the QoS settings.

### 31.5 Data Handling Optimizations

#### 31.5.1 Using Servants Directly

The simple example used the CORBA interfaces for writing and reading the data. Because the Data Writer and Reader servants are local to our process, TAO’s collocation optimizations allow it to very efficiently execute these operations without incurring many of the overheads of CORBA calls including marshaling and system call overhead.

A further increase in speed can be obtained by enabling TAO’s direct collocation stubs, but an even better approach is to directly access the servant that incarnates the CORBA object and make direct C++ calls on the servant. We saw an example of this when we accessed the publisher’s servant in order to attach the transport to the publisher. The `reference_to_servant()` template function was used to make this conversion and a similar technique is possible for Data Readers, Data Writers, and other DDS entities.

Here is the original subscriber code, modified to directly use the Data Writer servant (changes marked in **bold**).

```c++
void DataReaderListenerImpl::on_data_available(DDS::DataReader_ptr reader)
    throw (CORBA::SystemException)
{
    num_reads_ ++;
    try {
        MessageDataReader_var message_dr = MessageDataReader::_narrow(reader);
        if (CORBA::is_nil (message_dr.in ())) {
            std::cerr << "read: _narrow failed." << std::endl;
            return;
        }
        MessageDataReaderImpl* dr_servant =
            reference_to_servant< MessageDataReader_ptr,
            MessageDataReaderImpl>(message_dr.in());
        Messenger::Message message;
        DDS::SampleInfo si ;
        DDS::ReturnCode_t status = dr_servant->take_next_sample(message, si) ;
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
Similar modifications are possible for the Data Writer.

### 31.5.2 Other Reading Techniques

The DDS specification supplies a number of operations for reading and writing data samples. In the examples above we used the `take_next_sample()` operation, to read the next sample and “take” ownership of it from the reader. The Message Data Writer also has the following take operations.

- `take()` — Take a sequence of up to `max_samples` values from the reader
- `take_instance()` — Take a sequence of values for a specified instance
- `take_next_instance()` — Take a sequence of samples belonging to the same instance, without specifying the instance.

There are also “read” operations corresponding to each of these “take” operations that obtain the same values, but leave the samples in the reader and simply mark them as read in the `SampleInfo`.

Since these other operations read a sequence of values, they are more efficient when samples are quickly arriving. Here is a sample call to `take()` that reads up to 5 samples at a time.

```cpp
MessageSeq messages(5);
DDS::SampleInfoSeq sampleInfos(5);
DDS::ReturnCode_t status = message_dr->take(messages, sampleInfos, 5,
                                          DDS::ANY_SAMPLE_STATE,
                                          DDS::ANY_VIEW_STATE,
                                          DDS::ANY_INSTANCE_STATE);
```

The three state parameters specialize which samples are returned from the reader. See the DDS specification for details on their usage.

### 31.6 QoS Policies

The above examples use default QoS policies for the various entities. This section discusses which QoS policies are implemented in TAO’s DDS and the details of their usage.
31.6 QoS Policies

31.6.1 Supported Policies

Listed below are the QoS policies that are currently supported by TAO’s DDS. Any policy not listed here uses its default value. The default values of unsupported policies are as described in the DDS specification and some are discussed in 31.6.2.

Each policy defines a structure to specify its data. Each entity supports a subset of the policies and defines a QoS structure that is composed of the supported policy structures. The set of allowable policies for a given entity is constrained by the policy structures nested in its QoS structure. For example the Publisher’s QoS structure is defined in the specification’s IDL as follows.

```idl
module DDS {
    struct PublisherQos {
        PresentationQosPolicy presentation;
        PartitionQosPolicy partition;
        GroupDataQosPolicy group_data;
        EntityFactoryQosPolicy entity_factory;
    };
}
```

Setting policies is as simple as obtaining a structure with the default values set, modifying the individual policy structures as necessary, and then applying the QoS structure to an entity (usually when it is created).

**Note** TAO’s DDS implementation does not currently support changing the QoS of an existing entity, so all QoS policies must be applied when the entity is created.

31.6.1.1 Liveliness

The Liveliness policy applies to the Topic, Data Reader, and Data Writer entities via the `liveliness` member of their respective QoS structures. Below is the IDL related to the Liveliness QoS policy:

```idl
enum LivelinessQosPolicyKind {
    AUTOMATIC_LIVELINESS_QOS,
    MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
    MANUAL_BY_TOPIC_LIVELINESS_QOS
};

struct LivelinessQosPolicy {
```
The Liveliness policy controls when and how the service determines whether participants are alive, meaning they are still reachable and active. The `kind` member is restricted to Automatic (`AUTOMATIC_LIVENESS_QOS`) in TAO’s DDS implementation. This means that the service periodically polls participants for liveliness. The `lease_duration` member is set to the desired heartbeat interval. The default lease duration is a pre-defined infinite value, which disables any liveliness testing.

Data writers specify their own liveliness criteria and data readers specify the desired liveliness of their writers. Writers that can’t be contacted within the lease duration cause notification via the Data Reader Listener’s `on_liveliness_changed()` operation. Because TAO’s Liveliness is always set to Automatic, the `on_liveliness_changed()` callback is never called on the publisher side.

### 31.6.1.2 Reliability

The Reliability policy applies to the Topic, Data Reader, and Data Writer entities via the `reliability` member of their respective QoS structures. Below is the IDL related to the Reliability QoS policy:

```idl
enum ReliabilityQosPolicyKind {
  BEST_EFFORT_RELIABILITY_QOS,
  RELIABLE_RELIABILITY_QOS
};

struct ReliabilityQosPolicy {
  ReliabilityQosPolicyKind kind;
  Duration_t max_blocking_time;
};
```

This policy controls how data readers and writers treat the data samples they process. The Best Effort value (`BEST_EFFORT_RELIABILITY_QOS`) makes no promises as to the reliability of the samples and could be expected to drop samples under some circumstances. The Reliable value (`RELIABLE_RELIABILITY_QOS`) indicates that the service should eventually deliver all values to eligible Data Readers.

The Simple TCP transport supports the Reliable value for this policy and the Simple UDP transport only supports the Best Effort value. The
max\_blocking\_time member of this policy is used when the History QoS Policy is set to Keep All and the writer is unable to return because of resource limits. When this situation occurs and the writer blocks for more than the specified time, then the write fails with a timeout return code. The default for this policy is Best Effort

### 31.6.1.3 History

The History policy determines how samples are held in the Data Writer and Data Reader for a particular instance. For Data Writers these values are held until the Publisher retrieves them and successfully sends them to all connected subscribers. For Data Readers these values are held until “taken” by the application. This policy applies to the Topic, Data Reader, and Data Writer entities via the history member of their respective QoS structures. Below is the IDL related to the History QoS policy:

```idl
eenum HistoryQosPolicyKind {
    KEEP_LAST_HISTORY_QOS,
    KEEP_ALL_HISTORY_QOS
};

struct HistoryQosPolicy {
    HistoryQosPolicyKind kind;
    long depth;
};
```

The Keep All value (KEEP\_ALL\_HISTORY\_QOS) specifies that all possible samples for that instance should be kept. When Keep All is specified and the number of unread samples is equal to the Resource Limits field of max\_samples\_per\_instance then any incoming samples are rejected.

The Keep Last value (KEEP\_LAST\_HISTORY\_QOS) specifies that only the last depth values should be kept. When a data reader contains depth values, any incoming samples are kept and the oldest sample is discarded. This policy defaults to a Keep Last with a depth of one.

### 31.6.1.4 Resource Limits

The Resource Limits policy determines the amount of resources the service can consume in order to meet the requested QoS. This policy applies to the Topic, Data Reader, and Data Writer entities via the resource\_limits member of their respective QoS structures. Below is the IDL related to the Resource Limits QoS policy.
The `max_samples` member specifies the maximum number of samples a single Data Writer or Data Reader can manage across all of its instances. The `max_instances` member specifies the maximum number of instances that a Data Writer or Data Reader can manage. The `max_samples_per_instance` member specifies the maximum number of samples that can be managed for an individual instance in a single Data Writer or Reader. All of these members default to unlimited (DDS::LENGTH_UNLIMITED).

### 31.6.2 Unsupported Policies

The unsupported policies cannot be modified with TAO DDS and always take the default value. The following subsections discuss some of the default values that may affect application behavior.

#### 31.6.2.1 Entity Factory

The Entity Factory policy controls whether created entities are automatically enabled. The default is that all entities are automatically enabled on creation.

#### 31.6.2.2 Durability

The Durability policy controls whether Data Writers should maintain samples after they have been sent to known subscribers. By default the Durability is Volatile which means samples are discarded after being sent to all known subscribers. A side effect of this is that subscribers cannot recover samples sent before they connect.

#### 31.6.2.3 Presentation

The Presentation policy controls the ordering or grouping of samples in a topic. The default value of Instance means that all samples within an instance are delivered in the order the subscriber receives them. Samples from different instances of the same topic may be arbitrarily reordered by the service.
31.6.3 Policy Example

Here are some policies being set and applied for a publisher.

```cpp
DDS::DataWriterQos dw_qos;
pub->get_default_datawriter_qos (dw_qos);

dw_qos.history.kind = DDS::KEEP_ALL_HISTORY_QOS;

dw_qos.reliability.kind = DDS::RELIABLE_RELIABILITY_QOS;
dw_qos.reliability.max_blocking_time.sec = 10;
dw_qos.reliability.max_blocking_time.nanosec = 0;

dw_qos.resource_limits.max_samples_per_instance = 100;

DDS::DataWriter_var dw =
    pub->create_datawriter(topic.in (),
                            dw_qos,
                            DDS::DataWriterListener::_nil());
```

This code creates a publisher with the following qualities:

- History set to Keep All
- Reliability set to Reliable with a maximum blocking time of 10 seconds
- The maximum samples per instance resource limit set to 100

This means that when 100 samples are waiting to be delivered, the writer can block up to 10 seconds before returning an error code. These same QoS settings on the Data Reader side would mean that up to 100 unread samples are queued by the framework before any are rejected. Rejected samples are dropped and the SampleRejectedStatus is updated.

31.7 Pluggable Transport Details

The previous examples used the default configuration of the Simple TCP transport. In this section, we will show how to use the Simple UDP transport and how we can configure the different transports.

31.7.1 Using the Simple UDP Transport

Using the UDP transport involves the same steps as we have seen before of registering our transport type, creating a Transport Implementation, configuring the transport, and attaching it to the Publisher and Subscriber
servants. One difference is that we need to explicitly specify the address for
the UDP transport. Here is the code to create and configure the Simple UDP
Transport Implementation.

```cpp
TheTransportFactory->register_type(SIMPLE_UDP,
    new TAO::DCPS::SimpleUdpFactory());
TAO::DCPS::TransportImpl_rch udp_impl =
    TheTransportFactory->create(PUB_TRAFFIC_UDP, SIMPLE_UDP);

TAO::DCPS::SimpleUdpConfiguration_rch config =
    new TAO::DCPS::SimpleUdpConfiguration();
ACE_INET_Addr address ("localhost:16701");
config->local_address_ = address;

if (udp_impl->configure(config.in()) != 0) {
    std::cerr << "Failed to configure transport" << std::endl;
    return 1;
}
TAO::DCPS::AttachStatus status = pub_impl->attach_transport(udp_impl.in());
```

This configures the publisher to use the Simple UDP transport with port 16701
on the localhost interface assigned to this publisher. Configuring the
subscriber (or other publishers) should be identical to above but with different
addresses/ports assigned to each Transport Impl.

Because the Simple UDP transport does not support fragmentation of a single
sample into multiple packets, it currently limits the size of samples to about 64
KB. Attempting to send a sample over 64 KB with the Simple UDP transport,
results in an error message and the sample not being sent.

### 31.7.2 Configuring the Simple Transports

Currently, the only configuration possible for the Simple TCP and UDP
transports is the address assignment. This is discussed for UDP in the
preceding section and is identical in nature for the TCP transport. The TCP
transport defaults to using a random port number. You may need to explicitly
set the address if you have a multiple NICs or you wish to specify the port
number. The Simple TCP and UDP configuration objects also share a number
of parameters from the configuration base class, but your application code
should have no reason to modify their default values.
31.8 Using Built-In Topics

The built-in topics are published by the DCPSInfoRepo server whenever the -NOBITTS option is not specified. Four separate topics are defined for each domain that this server manages. Each is dedicated to a particular entity (Domain Participant, Topic, Data Writer, Data Reader) and publishes instances describing the state for each entity in the domain.

Subscriptions to the built-in topics are automatically created for each domain participant. To view the data you must simply obtain the built-in Subscriber and then use it to get the Data Reader for the built-in topic of your interest. Then the Data Reader can be used like any other Data Reader.

The next four sections detail the data published for each of the four built-in topics. Following these sections is some example code that shows how to read from a built-in topic.

31.8.1 DCPSParticipant Topic

The DCPSParticipant topic publishes information about the Domain Participants of the Domain. Here is the IDL that defines the structure published for this topic:

```
struct ParticipantBuiltinTopicData {
   _builtinTopicKey_t key;
    UserDataQosPolicy user_data;
};
```

Each Domain Participant is defined by a unique key, making each one its own instance within this topic.

31.8.2 DCPSTopic Topic

The DCPSTopic topic publishes information about the Topics in the Domain. Here is the IDL that defines the structure published for this topic:

```
struct TopicBuiltinTopicData {
    _builtinTopicKey_t key;
    string name;
    string type_name;
    DurabilityQosPolicy durability;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy liveliness;
};
```
Each Topic is identified by a unique key and is its own instance within this built-in topic. The members above identify the name of the Topic, the name of the topic type, and the set of QoS policies for that Topic.

### 31.8.3 DCPSPublication Topic

The DCPSPublication topic publishes information about the Data Writers in the Domain. Here is the IDL that defines the structure published for this topic:

```
struct PublicationBuiltinTopicData {
    BuiltinTopicKey_t key;
    BuiltinTopicKey_t participant_key;
    string topic_name;
    string type_name;
    DurabilityQosPolicy durability;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy liveliness;
    ReliabilityQosPolicy reliability;
    LifespanQosPolicy lifespan;
    UserDataQosPolicy user_data;
    OwnershipStrengthQosPolicy ownership_strength;
    PresentationQosPolicy presentation;
    PartitionQosPolicy partition;
    TopicDataQosPolicy topic_data;
    GroupDataQosPolicy group_data;
};
```

Each Data Writer is assigned a unique key when it is created and defines its own instance within this topic. The fields above identify the Domain Participant (via its key) that the Data Writer belongs to, the Topic name and type, and the various QoS policies applied to the Data Writer.
31.8.4 DCPSSubscription Topic
The DCPSSubscription topic publishes information about the Data Readers in the Domain. Here is the IDL that defines the structure published for this topic:

```c
struct SubscriptionBuiltinTopicData {
    BuiltinTopicKey_t key;
    BuiltinTopicKey_t participant_key;
    string topic_name;
    string type_name;
    DurabilityQosPolicy durability;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy latency_budget;
    LivelinessQosPolicy liveliness;
    ReliabilityQosPolicy reliability;
    DestinationOrderQosPolicy destination_order;
    UserDataQosPolicy user_data;
    TimeBasedFilterQosPolicy time_based_filter;
    PresentationQosPolicy presentation;
    PartitionQosPolicy partition;
    TopicDataQosPolicy topic_data;
    GroupDataQosPolicy group_data;
};
```

Each Data Reader is assigned a unique key when it is created and defines its own instance within this topic. The fields above identify the Domain Participant (via its key) that the Data Reader belongs to, the Topic name and type, and the various QoS policies applied to the Data Reader.

31.8.5 Built-In Topic Subscription Example
The following code uses a domain participant to get the built-in subscriber. It then uses the subscriber to get the Data Reader for the DCPSParticipant topic and subsequently reads samples for that reader.

```c
Subscriber_var bit_subscriber = participant->get_builtin_subscriber();
DDS::DataReader_var dr =
    bit_subscriber->lookup_datareader(BUILT_IN_PARTICIPANT_TOPIC);
DDS::ParticipantBuiltinTopicDataDataReader_var part_dr =
    DDS::ParticipantBuiltinTopicDataDataReader::_narrow(dr.in());

DDS::ParticipantBuiltinTopicDataSeq part_data;
DDS::SampleInfoSeq infos;
DDS::ReturnCode_t ret = part_dr->read ( part_data, infos, 20,
    DDS::ANY_SAMPLE_STATE,
    DDS::ANY_VIEW_STATE,
    DDS::ANY_INSTANCE_STATE) ;
```
The code for the other built-in topics is similar.

### 31.9 DCPS Service Participant Options

The options described in this section are passed to the service participant singleton when initializing the domain participant factory. Typically, the command line arguments are passed through as well, so these options can be specified on the command line of most DCPS-participating processes. We did this in the preceding examples by using the `TheParticipantFactoryWithArgs` macro:

```cpp
#include <dds/DCPS/Service_Participant.h>

int main (int argc, char* argv[])
{
    ::DDS::DomainParticipantFactory_var dpf =
        TheParticipantFactoryWithArgs(argc, argv);
}
```

The table below summarizes the DCPS service participant options.

**Table 31-1 DCPS Participant Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DCPSDebugLevel n</td>
<td>Integer value that controls the amount of debug information that service prints</td>
<td>0</td>
</tr>
<tr>
<td>-DCPSInfo objref</td>
<td>Object reference for locating the DCPS Information Repository</td>
<td>file://repo.ior</td>
</tr>
<tr>
<td>-DCPSChunks n</td>
<td>Configurable number of chunks that a data writer’s and reader’s cached allocators need to allocate when the resource limits are infinite</td>
<td>20</td>
</tr>
</tbody>
</table>

The debug level is useful for diagnosing issues with processes interacting with DDS. A debug level of 10 gives the maximum amount of debug information.

The `-DCPSInfo` option’s value is passed to `ORB::string_to_object()` and can be of any CORBA URL type understandable by TAO (`file`, `IOR`, `corbaloc`, `corbaname`).
The -DCPSChunks option allows application developers to tune the amount of memory pre-allocated when the RESOURCE_LIMITS are set to infinite. Once the allocated memory is exhausted, additional chunks are allocated/deallocated from the heap.

The Service_Participant class also provides methods that allow an application to configure the DDS service. See DDS/DCPS/Service_Participant.h for details.

### 31.10 dcps_ts.pl Command Line Options

The dcps_ts.pl script is located in $DDS_ROOT/bin and parses a single IDL file for DCPS-enabled types then generates type support code for those types. For each type it finds, such as xyz, it generates three files: xyzTypeSupport.idl, xyzTypeSupportImpl.h, and xyzTypeSupportImpl.cpp. Because a single IDL file could find multiple DCPS-enable types, an invocation of the script may generate a multitude of files. In the typical usage, the script is passed a number of options and the IDL file name as a parameter. For example,

$DDS_ROOT/bin/dcps_ts.pl --module=Mine Foo.idl

The following table summarizes the entire set of options the script supports. Note that many have terse and verbose variants of the same option.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--verbose</td>
<td>Enables/disables verbose execution</td>
<td>Quiet execution</td>
</tr>
<tr>
<td>--noverbose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--debug -d</td>
<td>Enable debug statements in the script</td>
<td>No debug output</td>
</tr>
<tr>
<td>--help -h</td>
<td>Prints a usage message and exits</td>
<td>N/A</td>
</tr>
<tr>
<td>--man</td>
<td>Prints a man page and exits</td>
<td>N/A</td>
</tr>
<tr>
<td>--dir=dirpath -S=dirpath</td>
<td>Subdirectory where IDL file is located</td>
<td>No subdir used</td>
</tr>
<tr>
<td>--export=macro -X=macro</td>
<td>Export macro used for generating C++ implementation code.</td>
<td>No export macro used</td>
</tr>
<tr>
<td>--pch=file</td>
<td>Pre-compiled header file to include in generated C++ files</td>
<td>No pre-compiled header included</td>
</tr>
</tbody>
</table>
Data Distribution Service

Table 31-2 dcps_ts.pl Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--module=mod</td>
<td>IDL module where type support code is located. Also used as the C++ namespace of the generated servants.</td>
<td>Interfaces and servants are in the global scope</td>
</tr>
<tr>
<td>-M=mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--timestamp -t</td>
<td>Backup any previously existing generated files with a timestamp suffix.</td>
<td>Old files are not backed up</td>
</tr>
<tr>
<td>--nobackup</td>
<td>Do not back up the previously generated files</td>
<td>Old files are not backed up</td>
</tr>
<tr>
<td>--idl=file</td>
<td>The IDL file to process.</td>
<td>IDL file is assumed to be a parameter</td>
</tr>
</tbody>
</table>

These options mainly divide into two main categories, those related to the execution of the script and those that control the generated code. In the former category are documentation options like --help and --man as well as script debugging options like --verbose and --debug.

The code generation options allow the application developer to use the generated code in a wide variety of environments. The -dir option lets you operate on IDL files in other directories and causes the generated IDL code to use the proper paths in the includes. The --export option lets you add an export macro to your class definitions. This is required if you plan on using the generated code in a library with a compiler (such as Visual C++) that uses the export keyword. The --pch option is required if the generated servant code is to be used in a component that uses precompiled headers. The --module option allows you to put the generated code in a C++ namespace and avoid name collisions and pollution of the global name space. The --timestamp and --nobackup options control whether older versions of the generated files are preserved with timestamp-appended file name or whether they are simply overwritten.

The --idl option allows you to specify IDL file to process with an option instead of with the simple parameter.
31.11 DCPS Information Repository Options

The table below shows the command line options when running the DCPSInfoRepo server.

Table 31-3 DCPS Information Repository Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o file</td>
<td>Write the IOR of the DCPSInfo object to the specified file</td>
<td>repo.ior</td>
</tr>
<tr>
<td>-d file</td>
<td>Load domain IDs from the specified file</td>
<td>domain_ids</td>
</tr>
<tr>
<td>-NOBITS</td>
<td>Disable the publication of built-in topics</td>
<td>Built-in topics are published</td>
</tr>
<tr>
<td>-a address</td>
<td>Listening address for built-in topics (when built-in topics are published).</td>
<td>Random port</td>
</tr>
<tr>
<td>-z</td>
<td>Turn on verbose transport logging</td>
<td>Minimal transport logging.</td>
</tr>
<tr>
<td>-?</td>
<td>Display the command line usage and exit</td>
<td>N/A</td>
</tr>
</tbody>
</table>

DDS clients usually use the IOR file that DCPSInfoRepo outputs to locate the service. The -o option allows you to place the IOR file into an application-specific directory or file name.

The domain file is a simple list of integer domain ID values, one per line. Any domain IDs used that do not appear in this file are assumed to be invalid and result in an INVALID_DOMAIN exception being thrown.

Applications that do not use built-in topics may want to disable them with -NOBITS to reduce the load on the server. If you are publishing the built-in topics, then the -a option lets you pick the listen address of the Simple TCP transport that is used for these topics.

Using the -z option causes the invocation of many transport-level debug messages. This option is only effective when the DCPS library is built with the DCPS_TRANS_VERBOSE_DEBUG environment variable defined.
Part 5

CIAO
CHAPTER 32

CIAO and CCM

32.1 Introduction

The OMG CORBA Component Model (CCM) (OMG Document formal/02-06-65) defines a specification for implementing component middleware. The Component-Integrated ACE ORB (CIAO) is TAO’s implementation of the CCM specification.

The CORBA Component Model is a step in the longtime evolution of software engineering best practices towards higher levels of abstraction. CCM is a realization of the concept of composing software from reusable, pluggable components, assembled into an application at run time. When properly applied, component-based software development promotes improved software reuse, deployment flexibility, and programmer productivity.

32.1.1 Prerequisites

To better understand this chapter, the reader should be familiar with the content of the following chapters from this guide:

- Chapter 2, “Building ACE and TAO”
32.1.2 What is a Component?

A component is a pluggable, self-contained software entity consisting of its own encapsulated business logic and data with clearly defined interfaces for collaboration. A component defines both the capabilities it provides and the services it requires as well as events it publishes and consumes, as illustrated by the diagram.

![Diagram of a CCM Component](image)

**Figure 32-1 A CCM Component**

The CORBA 2.x object model lacks the expressiveness required to create pluggable components. A CORBA 2.x IDL interface specifies a contract between a client and a server. That contract specifies what the server provides and what the client can expect. However, a great deal of information is missing from that IDL contract. A client or server has no formal mechanism to specify what it requires—namely, which IDL interfaces it depends upon to accomplish its tasks. These dependencies are hidden in the implementation code. Without knowledge of what each client or server requires, it is impossible to connect the clients and servers at run time in a generic way.
The CORBA Component Model includes new IDL constructs for expressing both the client and the server sides of component collaboration. This new edition of IDL is called IDL3. IDL3 is a superset of traditional CORBA IDL, or IDL2. The TAO 1.4a IDL compiler accepts both IDL3 and IDL2 interface specifications.

A component defines its collaborations in terms of provided and required interfaces. An IDL3 component specification consists of ports that indicate how the component interacts with other components as both a client and a server. There are several types of ports providing various capabilities:

- A facet defines an IDL interface provided by a component. This is the server-side of the traditional IDL contract.
- A receptacle defines an IDL interface that is used by a component. The component may interact with that interface either through synchronous calls or through AMI. Facets and receptacles are connected via assembly descriptors that are processed at run-time.
- An event source or publisher defines an event type that is published by a component. CCM events are strongly typed, as our example will illustrate.
- An event sink or consumer defines an event type that is consumed by a component. Event sources and sinks are connected via assembly descriptors that are processed at run-time.
- An attribute provides a mechanism for configuring component properties at application start-up.

An application consists of several components packaged together and deployed at run time. A CCM-based application may consist of numerous binary component implementations implemented in several different programming languages communicating through CORBA.

The CCM specification defines a Component Implementation Framework (CIF) consisting of tools to simplify the implementation of components. The CIF uses the Component Implementation Definition Language (CIDL), through which a component developer defines a composition to describe a component’s implementation details. A CIDL compiler generates a skeletal version of the component’s C++ implementation, or executor. The developer is left to concentrate on application logic.
### 32.1.3 Component Deployment

A developer-configures an application’s component connections—facet to receptacle, event source to event sink—via descriptor files that a component server process loads at run time. The component server creates a component container to instantiate a component and connect it to any collaborating components through the appropriate ports. The component itself is deployed in a library that is dynamically loaded into the component server at run time.

CORBA is the underlying middleware infrastructure for the component containers. The container programming model is built on the Portable Object Adapter (POA). Components communicate through CORBA, assuring interoperability. The diagram illustrates the component container’s relationship to the CORBA infrastructure.

![Diagram of Component Container and CORBA Infrastructure](image)

**Figure 32-2 The Component Container and the CORBA Infrastructure**

### 32.1.4 Summary of the CCM Programming Model

The CCM model of component programming extends the CORBA 2.x programming model in the following ways:

- A component specifies not only what it provides, but also what it requires.
- A component can provide multiple interfaces that are not related through inheritance.
32.1 Introduction

- A component specifies events it publishes and consumes directly in its interface. Events are strongly typed value objects.
- An application developer assembles and deploys a component-based application by writing standard XML-based assembly and deployment descriptors. The component server reads the descriptors at run-time to load libraries and connect components, promoting loose coupling of component implementations.
- A component developer can add capabilities to an existing component without affecting existing clients by providing a new facet.
- A component developer does not need to have any direct interaction with the Portable Object Adapter. The component container interacts with the POA.
- A component developer does not write a `main()` function.
- The component container instantiates and destroys the component.
- The component server provides standard services such as event publication, transactions, persistent state, and security and enforces usage policies consistently.

A CCM client does not have to be component-aware. A CORBA 2.x client can bind to a component facet and interact with it without any knowledge that it is part of a CCM component.
32.1.5 Road Map

The following sections illustrate the CCM programming model with an example. The example illustrates the steps involved in developing a CIAO application by tracing the road map outlined in the diagram.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-3 Road Map

As you can see, component development and deployment primarily consists of five phases: defining interfaces, implementing interfaces, describing the deployment, building, and running. Defining and implementing interfaces should be familiar to any CORBA developer. We will find that some of the steps in implementing an IDL3 interface are a bit different as we take advantage of the CCM programming model. Describing the deployment involves defining XML descriptors to define how each component is composed from its libraries and how the components are connected together to form an application. In the fourth step, building, we create a set of dynamic libraries for each component. Finally, we run the application by executing component servers to load the dynamic libraries and connect the components together.
32.2 Example - The Messenger Application

Our CIAO example builds on the Messenger example used throughout this guide. The example’s source code, build files, and XML descriptor files are in the $TAO_ROOT/DevGuideExamples/CIAO/Messenger directory.

The CIAO Messenger example consists of three components: a Messenger, a Receiver, and an Administrator. The Messenger publishes message events and provides a history of all published messages. The Receiver subscribes to message events and retrieves the Messenger’s message history. The Administrator controls the Messenger, starting and stopping publication and changing the attributes of what the Messenger publishes. The relationship between the three component types is demonstrated by the following component diagram:

![Figure 32-4 Messenger Component Diagram](image)

The diagram illustrates that the Messenger component provides three facets, Runnable, Publication, and History. Each facet is an IDL interface. The Messenger component also publishes Message events. Each Message event is a value-based event type. The Receiver component has a receptacle that
connects to the Messenger’s History facet. It also consumes Message events published by the Messenger. Finally, the Administrator component has two receptacles connected to the Messenger’s Runnable and Publication facets, respectively.

The Messenger does not start publishing messages immediately at start-up. The Administrator connects to the Messenger’s Runnable facet and invokes the start() operation on it to trigger message publication. Upon receiving a start() request, the Messenger publishes messages to all connected Receivers until the Administrator tells it to stop(). The “start publication” collaboration is illustrated in the following interaction diagram:

![Interaction Diagram](image)

**Figure 32-5 Start Message Publication**

### 32.2.1 The Messenger Application’s IDL Interfaces

The first task is to specify the Messenger application’s interfaces using IDL. Or, more accurately, using IDL3. Next, we create a component type specification for the Messenger, Receiver, and Administrator. After that, we specify standard IDL interfaces for each of the facets provided by the
Messenger, namely the History, Runnable, and Publication. Finally, we create a Message event type whose instances the Messenger publishes.

**Define an IDL interface for each component and its facets**
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-6 Road Map

### 32.2.1.1 The Messenger Component and Facets

The Messenger component provides facets that implement the Runnable, Publication, and History interfaces. It also publishes a Message. Each Receiver component consumes the Messages published by the Messenger and uses the History facet provided by the Messenger. The Administrator component uses the Runnable and Publication facets provided by the Messenger.

First, we specify IDL interfaces for the Runnable and Publication facets. Both of these are IDL2 interfaces that would be recognized by any CORBA client:

```c++
// file Runnable.idl
interface Runnable {
    void start();
    void stop();
};

// file Publication.idl
interface Publication {
    attribute string text;
    attribute unsigned short period;
};
```
We put each IDL interface in its own file as a programming convention. The Runnable interface provides control over starting and stopping of message publication. The Publication interface provides control over the published message text and the period, in seconds, between messages.

The Messenger component publishes Message events. We define the Message type using the new IDL3 keyword eventtype. An eventtype is an IDL value type that inherits from the abstract value type Components::EventBase. Our Message event type has three public string members: subject, user, and text.

```
// file Message.idl
#include <Components.idl>

eventtype Message {
    public string subject;
    public string user;
    public string text;
};
typedef sequence<Message> Messages;
```

We must include the IDL file Components.idl to use IDL3 keywords such as eventtype. Like any IDL value type, the Message event type may contain operations and a factory. For more information on value types, see Chapter 11.

However, we can simplify our event type implementation by restricting the contents of the event type to public data members. For such an event type, the IDL compiler generates a full event type implementation and automatically registers the event type factory for us. Therefore, we do not add operations or a factory to the event type.

The History facet contains operations to retrieve published Message events.

```
// file History.idl
#include <Components.idl>
#include <Message.idl>

interface History {
    Messages get_all();
    Message get_latest();
};
```
The implementation of the History facet must keep track of each message that it publishes for later retrieval by clients.

Finally, we declare the Messenger component. The Messenger declaration illustrates several of the new IDL3 keywords introduced for component-based programming.

```
// file Messenger.idl

#include <Components.idl>
#include <Runnable.idl>
#include <Publication.idl>
#include <Message.idl>
#include <History.idl>

component Messenger {
    attribute string subject;

    provides Runnable control;
    provides Publication content;

    publishes Message message_publisher;
    provides History message_history;
};

devicetype MessengerHome manages Messenger {};
```

The Messenger’s component specification must include Components.idl to make the IDL3 keywords available. It also includes IDL files for each of its three facets and for the Message events it publishes.

The keyword `component` is a new IDL3 keyword that is used to define a component.

```
component Messenger {
    
    The component’s definition can contain IDL attributes just like an IDL2 interface. However, the component’s definition may not contain IDL operations.

    The Messenger component contains one attribute, the subject.

    attribute string subject;
```
Despite of the fact that the subject attribute is writable it is not exposed to the Messenger’s clients.

The Messenger component provides three facets.

```java
provides Runnable control;
provides Publication content;
provides History message_history;
```

Each facet is an IDL interface. A component uses the provides keyword to indicate the services that it offers. In the example, the Messenger’s three facets are a Runnable facet called control for starting and stopping message publication, a Publication facet called content for control over the message content and publication period, and a History facet called message_history for access to all messages published by the component. There is no limit to the number of clients that may access the Messenger’s facets.

The Messenger component publishes events:

```java
publishes Message message_publisher;
```

Recall that a Message is an event type. Published events are strongly typed. There is no limit to the number of subscribers for a published event. The Messenger component has neither direct knowledge of the event’s subscribers nor knowledge of the underlying messaging mechanism.

A publishes port may publish to an unlimited number of subscribers. A second kind of publisher, called an emitter, is limited to one subscriber. An emitter uses the emits keyword instead of the publishes keyword. The CCM deployment framework enforces the emitter’s limitation to one subscriber at deployment time. Aside from the keyword, the emitter’s IDL syntax is the same as the publisher’s. For example:

```java
emits Message message_publisher;
```

The home, called MessengerHome, manages the life cycle of the component.

```java
home MessengerHome manages Messenger {};
```
Each component has a corresponding home. The component server uses the home to create and destroy component instances. Our Messenger’s home is the simplest possible home, implicitly defining a create() operation. The home construct will be discussed in more detail later.

### 32.2.1.2 The Receiver Component

The Receiver component receives Message events from the Messenger and retrieves the message History from the Messenger.

```idl
// file Receiver.idl
#include <Components.idl>
#include <Message.idl>
#include <History.idl>

cOMPONENT Receiver {
    consumes Message message_consumer;
    uses History message_history;
};

home ReceiverHome manages Receiver {};
```

The Receiver does not expose any facets, but instead indicates what it requires via a uses specification. The Receiver uses a History facet, and consumes Message events. The specification of not only what a component offers but also what it requires is a significant step forward, as it enables connection of components at deployment time. Both of these Receiver receptacles are connected to corresponding facets on the Messenger component at deployment.

The Receiver also has a home, ReceiverHome, which is responsible for creating and destroying Receiver component instances. Again, this is the simplest possible home declaration.

```idl
home ReceiverHome manages Receiver {};
```

---

**Note** The Receiver’s IDL file does not have a dependency on the Messenger. The Receiver knows about Message and History, but it does not need to know anything about the component that provides those services. A component may depend on IDL interfaces and event types, but it need not depend on other components.
32.2.1.3 The Administrator Component

Finally, the third component type, an Administrator, triggers the Messenger’s event publication and controls the period of its publication and the text that it publishes.

```idl
// file Administrator.idl
#include <Components.idl>
#include <Runnable.idl>
#include <Publication.idl>

component Administrator {
    uses multiple Runnable runnables;
    uses multiple Publication content;
};

home AdministratorHome manages Administrator {};
```

The Administrator uses both the Runnable and Publication facets provided by the Messenger. These two receptacles are later connected to corresponding facets provided by the Messenger. The Administrator’s home is responsible for creating and destroying the Administrator component instance at run time.

The uses multiple keyword on the Administrator’s runnables and content receptacles indicates that the Administrator can connect to more than one Runnable facet and more than one and Publication facet. These facets may be provided by the same component or by different components; the Administrator does not need to know. In our sample deployment the Administrator connects to one Runnable facet and one Publication facet, both from the same Messenger component.

---

**Note** The Administrator’s IDL file does not have a dependency on the Messenger. The Administrator knows about Runnable and Publication, but it does not need to know anything about the component that provides those services.

The Administrator, like all components, has a home to manage its life cycle:

```idl
home AdministratorHome manages Administrator {};
```
The homes in our example are the simplest possible. The default home contains a factory that acts like a default constructor. It is possible to override that factory and provide parameters to be passed into it.

To summarize, we’ve been exposed to several new IDL3 keywords:

**Table 32-1 IDL3 Keywords**

<table>
<thead>
<tr>
<th>IDL3 Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>Declares a component that can provide and use facets, publish and consume events</td>
</tr>
<tr>
<td>provides</td>
<td>Declares an IDL interface that the component offers; the interface defines a service offered</td>
</tr>
<tr>
<td>uses</td>
<td>Declares an IDL interface that the component requires</td>
</tr>
<tr>
<td>uses multiple</td>
<td>Declares that the component can connect to one or more instances of the required interface</td>
</tr>
<tr>
<td>eventtype</td>
<td>Declares an event type that the component publishes; the eventtype is an IDL valuetype</td>
</tr>
<tr>
<td>publishes</td>
<td>Declares that the component publishes instances of an event type to a potentially unlimited number of consumers</td>
</tr>
<tr>
<td>emits</td>
<td>Declares that the component publishes instances of an event type to exactly one consumer</td>
</tr>
<tr>
<td>consumes</td>
<td>Declares that the component expects the event type to be published to it by one or more publishers</td>
</tr>
<tr>
<td>home</td>
<td>Declares an interface used by the component container to manage the component’s life cycle</td>
</tr>
<tr>
<td>manages</td>
<td>Declares which component is managed by the home</td>
</tr>
</tbody>
</table>

### 32.2.2 Implementing the Components

The CORBA Component Model specification defines a Component Implementation Framework (CIF) consisting of tools to simplify and automate the implementation of components. A significant part of the CIF is the Component Implementation Definition Language (CIDL), through which a component developer provides implementation details for each component type. The CIDL compiler compiles the CIDL files and generates a significant portion of the C++ implementation code. The developer is left to concentrate on application logic.
We write CIDL files for the Messenger, Receiver, and Administrator component types. Each CIDL file contains a component composition.

```
- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
    - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application
```

**Figure 32-7 Road Map**

### 32.2.2.1 The Messenger Composition

The primary entity of a CIDL file is a composition. A composition describes how a component is connected to its home. A component can be instantiated by more than one home; the composition designates the home responsible for the component.

The declaration of the Messenger’s composition follows:

```c
// file Messenger.cidl
#include <Messenger.idl>

composition session Messenger_Impl
{
    home executor MessengerHome_Exec
    {
        implements MessengerHome;
        manages Messenger_Exec;
    };
};
```

The session is the component’s life cycle category. A session composition provides transient object references and maintains its transient state for the lifetime of the session. Once the component is destroyed, its object references
are invalidated and its state is lost. The other valid composition life cycle categories are entity, service, and process. They are discussed later.

The name of the composition is Messenger_Impl. The CIDL compiler generates its implementation code into a C++ namespace called Messenger_Impl. The composition can have any name; it is customary to end the name with Impl.

An implementation of a component or a home is called an executor. A CCM developer implements an executor rather than a servant. The CIDL compiler generates two abstract C++ executor classes, one for the component and one for its home, using the names Messenger Exec and MessengerHome Exec specified in the CIDL composition. The Messenger executors may have any name; it is customary to end the each with the suffix _Exec.

```cpp
home executor MessengerHome_Exec
{
    implements MessengerHome;
    manages Messenger_Exec;
}
```

The home executor defines which home is used to manage the life cycle of the Messenger component.

The implements declaration declares which of the component’s homes manages the component’s life cycle. The Messenger component only has one home, the MessengerHome, so that is the home we’ll use. Note that we don’t need to indicate that the MessengerHome manages the Messenger component; that relationship is defined in the MessengerHome’s declaration.

The component developer overrides pure virtual methods in the generated executor classes to provide the component implementation. The CIDL compiler can optionally generate a default implementation of each C++ executor class. By default, it appends _i to the executor class name. The default implementation of the Messenger executor is Messenger_exec_i, and the default implementation of the MessengerHome executor is MessengerHome_exec_i. The component developer fills the application logic into the generated Messenger executor implementation. The CIDL compiler generates a full implementation for the MessengerHome’s executor, so no developer intervention is required.
32.2.2.2 The Receiver and Administrator Compositions

The Receiver and Administrator compositions are similar to the Messenger composition.

// file Receiver.cidl
#include <Receiver.idl>

composition session Receiver_Impl
{
    home executor ReceiverHome_Exec
    {
        implements ReceiverHome;
        manages Receiver_Exec;
    }
};

The Receiver’s composition is called Receiver_Impl, and it’s home executor implements the ReceiverHome. CIDL compiler generates an abstract executor class for the ReceiverHome called ReceiverHome_Exec and an abstract executor class for the Receiver component called Receiver_Exec. Optionally, the CIDL compiler can generate default implementations of the two executors.

// file Administrator.cidl
#include <Administrator.idl>

composition session Administrator_Impl
{
    home executor AdministratorHome_Exec
    {
        implements AdministratorHome;
        manages Administrator_Exec;
    }
};

The Administrator’s composition is called Administrator_Impl, and its home executor implements the AdministratorHome. The CIDL compiler generates an abstract executor class for the AdministratorHome called AdministratorHome_Exec and an abstract executor class for the Administrator component called Administrator_Exec.
To summarize, we’ve been exposed to several new CIDL keywords:

**Table 32-2 CIDL Keywords**

<table>
<thead>
<tr>
<th>CIDL Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>composition</td>
<td>Declares a set of entities that work together to manage the component’s life cycle and implement the component’s behavior</td>
</tr>
<tr>
<td>session</td>
<td>A component category characterized by transient state and transient object identity</td>
</tr>
<tr>
<td>service</td>
<td>A component category characterized by objects having no duration beyond the lifetime of a single client interaction</td>
</tr>
<tr>
<td>entity</td>
<td>A component category characterized by persistent state that is visible to the user and persistent object identity</td>
</tr>
<tr>
<td>process</td>
<td>A component category characterized by persistent state that is not visible to the user and persistent object identity</td>
</tr>
<tr>
<td>executor</td>
<td>Declares the name of the abstract component home executor class</td>
</tr>
<tr>
<td>implements</td>
<td>Declares the home that manages the component</td>
</tr>
<tr>
<td>manages</td>
<td>Declares the name of the abstract component executor class</td>
</tr>
</tbody>
</table>

### 32.2.3 Compiling the IDL and CIDL

We compile the IDL files with TAO’s IDL compiler. The TAO 1.4a IDL compiler recognizes IDL3 constructs such as `component` and `eventtype`. Additional information on compiling the Messenger’s IDL files is contained in 32.2.6.

We compile the CIDL files with CIAO’s CIDL compiler. Additional information on compiling the Messenger’s CIDL files is also contained in 32.2.6. The CIDL Compiler Reference in 32.5 contains more extensive information on using the CIDL compiler.

This section concentrates on the output of the IDL and CIDL compilers rather than the mechanics of executing the IDL and CIDL compilers.

The CIDL compiler can generate most of the code for home, component, and facet executor implementations through its `--gen-exec-impl` command-line option. For each component, home, or facet it generates a C++ class that inherits from a generated abstract executor class, leaving the component developer to fill in the application logic.
The diagram shows the files that the CIDL compiler generates when it compiles `Messenger.cidl`.

![Diagram showing file generation](image)

**Figure 32-8 Running the CIDL Compiler**

We show both `Messenger.cidl` and `Messenger.idl` as inputs to the CIDL compiler because the `Messenger.cidl` file includes `Messenger.idl`. 
The CIDL compiler generates an IDL file, `MessengerE.idl`, containing local interfaces for the Messenger’s component, home, and facet executors. We compile this IDL file with the IDL compiler to generate an abstract C++ executor class for each component and facet. Each component, home, and facet executor implementation implements one of the local interfaces declared in `MessengerE.idl`.

The CIDL compiler also generates complete C++ header and implementation files for the servant classes. There is a servant class for each component, home, and facet executor class. The CCM developer does not directly instantiate servants; instead, the component container instantiates servants and registers them with the Portable Object Adapter automatically.

The CIDL compiler optionally generates default component, home, and facet executor implementation classes in files called `Messenger_exec.h` and `Messenger_exec.cpp`. Those files contain definitions for five classes: `Messenger_exec_i`, `MessengerHome_exec_i`, `Runnable_exec_i`, `Publication_exec_i`, and `History_exec_i`. The latter three classes are executors for the Messenger’s `Runnable`, `Publication`, and `History` facets. For safety, copy the `Messenger_exec.h` and `Messenger_exec.cpp` files to something like `Messenger_exec_i.h` and `Messenger_exec_i.cpp`. You may also want to break the implementations for `History_exec_i`, `Runnable_exec_i`, etc., into different header and implementation files as we’ve done in our sample code.
The diagram illustrates the Messenger executor’s classes.

![Diagram of Messenger Executor's Classes]

**Figure 32-9 The Messenger Executor’s Classes**

The table summarizes the Messenger’s executor implementation classes.

**Table 32-3 Executor Implementation Classes**

<table>
<thead>
<tr>
<th>Executor Implementation Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messenger_exec_i</td>
<td>Implements the Messenger component</td>
</tr>
<tr>
<td>MessengerHome_exec_i</td>
<td>Implements the MessengerHome</td>
</tr>
<tr>
<td>Runnable_exec_i</td>
<td>Implements the Runnable facet</td>
</tr>
<tr>
<td>Publication_exec_i</td>
<td>Implements the Publication facet</td>
</tr>
<tr>
<td>History_exec_i</td>
<td>Implements the History facet</td>
</tr>
</tbody>
</table>
32.2.4 Implementing the Executors

The CIAO CIDL compiler generates an empty implementation of each component and facet executor. In the following sections, we implement the facet executors for the Runnable, Publication, and History facets and the component executors for the Messenger, Receiver, and Administrator components.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-10 Road Map
32.2.4.1 The Runnable Facet Executor

The Runnable facet is provided by the Messenger component and permits a client to start and stop message publication. The component diagram highlights the role of the Messenger’s Runnable facet.

Recall that the Runnable IDL interface is as follows:

```c++
// file Runnable.idl
interface Runnable {
    void start();
    void stop();
};
```
The CIDL compiler generates a default Runnable executor with empty implementations of `start()` and `stop()`. The class diagram illustrates the Runnable executor’s class hierarchy.

![Class Diagram for the Runnable Executor](image)

**Figure 32-12 Class Diagram for the Runnable Executor**

The IDL compiler generates a Runnable stub. The CIDL compiler generates an abstract executor base class, `CCM_Runnable`, and optionally generates an empty executor implementation, `Runnable_exec_i`. The CIDL compiler generates a default constructor, a destructor, and a virtual method for each of Runnable’s IDL operations and attributes.
For each IDL interface “MyInterface” that is a facet of a component, the CIDL compiler generates an abstract facet executor class called “CCM_MyInterface.”

An executor is a local CORBA object. Its generated implementation class also inherits from a TAO-specific class called TAO_LoCal_ReferCted_object that marks Runnable_exec_i as a CORBA::LocalObject and provides reference counting that we’ll use in the Messenger executor implementation. Additional information on local objects can be found in Chapter 13.

The Messenger component only publishes messages when it can acquire the Runnable executor’s run_lock. If the Messenger cannot acquire the run lock, it blocks waiting for it to be released. A client of the Runnable facet controls the run lock via the start() and stop() operations.

The Runnable executor implementation follows. Changes to the CIDL-generated empty executor implementation are in bold:

```c++
// file Runnable_exec_i.h
#include "Messenger_serv.h"
#include "tao/LocalObject.h"
#include <ace/Synch.h>

namespace Messenger_Impl
{
  class MESSENGER_EXEC_Export Runnable_exec_i
   : public virtual ::CCM_Runnable,
     public virtual TAO_LoCal_ReferCted_object
  {
    public:
      Runnable_exec_i (void);
      virtual ~Runnable_exec_i (void);

      ACE_Mutex& get_run_lock();

    private:
      ACE_Mutex run_lock_;
  }
}
```
The included `Messenger_svnt.h` header file contains the servant class definitions for the `Runnable`, `Publication`, `History`, `Messenger`, and `MessengerHome`. A component developer does not implement servant classes; instead, the CIDL compiler generates servant classes and the component container automatically instantiates them at run time. A component developer implements executor classes that have no relationship to the server’s POA. The automatically-generated servant class delegates its execution to the developer-written local executor object.

The executor implementation inherits from the generated abstract executor base class, `CCM_Runnable`, and from `TAO_Local_RefCounted_Object`. `CCM_Runnable`, in turn, inherits from the generated `Runnable` stub class that the client uses. Thus, the executor implements the `Runnable` interface generated by the IDL compiler. The `start()` and `stop()` operations are declared as pure virtual methods in the `CCM_Runnable` class, forcing the executor to implement them.

Note the lack of inheritance from a `POA_Runnable` class; instead, the CIDL compiler generates a `Runnable_Servant` class for us.

The inheritance from `TAO_Local_RefCounted_Object` enforces two behaviors: first, the executor is a `CORBA::LocalObject`, meaning that it can only be used from within the server process; second the executor has reference counting, meaning that the inherited `_add_ref()` and `_remove_ref()` operations must be used to manage the executor’s memory.

Our `Runnable` implementation contains a private `ACE_Mutex` lock and a public accessor method to retrieve it. The `Messenger` acquires this `run_lock` before publishing each message and releases it after publishing each message. If the `Messenger` cannot acquire the `run_lock`, it blocks until the lock is released. A `Runnable` client can acquire and release the `run_lock` through the `start()` and `stop()` operations. In this way, a client can control whether or not the `Messenger` publishes any messages.

The CIDL compiler also generates an empty, default implementation of the `Runnable_exec_i` class. We implement a constructor, the `start()` and `stop()` operations, and an accessor for the mutex lock. Changes to the CIDL-generated default executor implementation code are in **bold**.
// file Runnable_exec_i.cpp

#include "Messenger_exec_i.h"
#include "ciao/CIAO_common.h"

namespace Messenger_Impl
{
    //==================================================================
    // Facet Executor Implementation Class:   Runnable_exec_i
    //==================================================================

    Runnable_exec_i::Runnable_exec_i (void)
    {
        // initially, the Messenger does not publish
        this->stop();
    }

    Runnable_exec_i::~Runnable_exec_i (void)
    {
    }

    // Operations from ::Runnable

    void
    Runnable_exec_i::start ()
    throw (CORBA::SystemException)
    {
        // Your code here.

        // allows the Messenger to acquire the lock and publish
        this->run_lock_.release();
    }

    void
    Runnable_exec_i::stop ()
    throw (CORBA::SystemException)
    {
        // Your code here.

        // prevents the Messenger from acquiring the lock; can’t publish
        this->run_lock_.acquire();
    }

    ACE_Mutex&
    Runnable_exec_i::get_run_lock()
    {
        return this->run_lock_;  
    }
}
The Runnable executor creates an ACE_Mutex lock for the Messenger to acquire in its event loop before publishing messages. If the Messenger can’t acquire the lock, then it does not publish messages. This agreement between the Runnable executor and the Messenger executor controls the suspension and resumption of message publication. Initially, the Runnable executor holds the lock. The implementations of start() and stop() release and acquire the lock, respectively. The get_run_lock() accessor exposes the lock to the Messenger executor.

The CIDL compiler also generates executor implementations for thePublication and History interfaces and the Messenger, Receiver, and Administrator components.

### 32.2.4.2 The Publication Facet Executor

The Publication facet is provided by the Messenger component and permits a client to modify the text published and the period (in seconds) between published messages.

Recall that the Publication IDL interface is as follows:

```cpp
interface Publication {
  attribute string text;
  attribute unsigned short period;
};
```

The CIDL compiler generates an empty implementation of the Publication executor. We add private class attributes to keep track of the message subject, text, and period. Changes to the CIDL-generated code are in **bold**.

```cpp
#include <string>
#include <ace/Synch.h>

namespace Messenger_Impl {
  class MESSENGER_EXEC_Export Publication_exec_i : public virtual ::CCM_Publication,
    public virtual TAO_Local_RefCounted_Object {
    public:
      Publication_exec_i (const char* text,
        CORBA::UShort period);
      virtual ~Publication_exec_i (void);

      // Operations from ::Publication
```
virtual char* text ()
    throw (CORBA::SystemException);

virtual void text (const char* text)
    throw (CORBA::SystemException);

virtual CORBA::UShort period ()
    throw (CORBA::SystemException);

virtual void period (CORBA::UShort period)
    throw (CORBA::SystemException);

private:
    std::string text_;  
    CORBA::UShort period_;  
    ACE_Mutex lock_;  
};
}

The pattern is similar to the Runnable executor’s. The Publication_exec_i executor inherits from both the generated CCM_Publication class and TAO’s TAO_LocalRefCounted_Object class. The accessor and modifier for the text and period attributes are declared as pure virtual methods in the CCM_Publication class, forcing us to implement them in our executor.

The text_ and period_ class members hold information about the publication. Because clients can modify the text and period, the executor uses an ACE_Mutex lock to protect them from simultaneous access. We have to assume that a provided facet might be accessed by multiple threads at the same time.

The implementation of the Publication executor follows. Again, changes to the CIDL-generated default executor implementation code are in **bold**.

```c++
#include "Publication_exec_i.h"
#include "ciao/CIAO_common.h"

namespace Messenger_Impl
{
    //===============================================================
    // Facet Executor Implementation Class:   Publication_exec_i
    //===============================================================

    Publication_exec_i::Publication_exec_i ( const char* text,
```
32.2 Example - The Messenger Application

```
CORBA::UShort period )
    : text_( text ),
    period_( period)
{
}

Publication_exec_i::~Publication_exec_i (void)
{
}

// Operations from ::Publication

cchar*
Publication_exec_i::text ()
    throw (CORBA::SystemException)
{
    ACE_Guard<ACE_Mutex> guard(this->lock_);
    return CORBA::string_dup( this->text_.c_str() );
}

void
Publication_exec_i::text (const cchar* text)
    throw (CORBA::SystemException)
{
    ACE_Guard<ACE_Mutex> guard(this->lock_);
    this->text_ = text;
    ACE_DEBUG((LM_INFO, ACE_TEXT("publication text changed to %s\n"), text ));
}

CORBA::UShort
Publication_exec_i::period ()
    throw (CORBA::SystemException)
{
    ACE_Guard<ACE_Mutex> guard(this->lock_);
    return this->period_;
}

void
Publication_exec_i::period (CORBA::UShort period)
    throw (CORBA::SystemException)
{
    ACE_Guard<ACE_Mutex> guard( this->lock_ );

    if ( period > 0 ) {
        this->period_ = period;
        ACE_DEBUG((LM_INFO,
            ACE_TEXT("publication period changed to %d seconds\n"), period ));
    } else {
        ACE_DEBUG((LM_INFO,
```
The Publication executor contains text and a publication period. Because the client may change either the text or publication period, we protect both with a mutex lock. The constructor sets the text and period values. The attribute accessors and modifiers are straightforward, protecting those values with the mutex lock. The period modifier ensures that the new period is a positive number.

32.2.4.3 The History Facet Executor

The Messenger component stores the messages that it publishes in a History executor. The History executor contains an STL list of published Message events. We protect access to the list with an ACE_Mutex lock because multiple threads might add to and query the History list simultaneously. We must assume that simultaneous access will happen.

Recall that the History IDL interface is as follows, where Message is an event type:

```cpp
#include <Message.idl>

interface History {
    Messages get_all();
    Message get_latest();
};
```

The CIDL-generated History executor implementation follows, with our changes in bold.

```cpp
#include "Messenger_svnt.h"
#include "Messenger_exec_export.h"
#include "tao/LocalObject.h"

#include <list>
#include <ace/Synch.h>

namespace Messenger_Impl {

    class MESSENGER_EXEC_Export History_exec_i : public virtual ::CCM_History,
    public virtual TAO_Local_RefCounted_Object
```
We add a mutex lock and an STL list of messages as private class attributes. The lock protects the message list from simultaneous access by multiple threads. The STL list stores Message_vars to properly handle reference counting and memory ownership. The Messenger component uses the public add() method to add messages to the history as it publishes them.

The History executor implementation follows. As always, changes to the CIDL-generated default executor implementation code are in bold. Comments are interspersed with the code.

```c++
namespace Messenger_Impl
{
    // Facet Executor Implementation Class: History_exec_i
    // ==============================================================

    History_exec_i::History_exec_i (void)
    {
    }

    History_exec_i::~History_exec_i (void)
    {
    }

    // Operations from ::History
```
The implementation of the history’s get_all() operation is the most challenging. It converts the STL list of Message_vars into an IDL sequence of Messages.

::Messages*
History_exec_i::get_all ()
throw (::CORBA::SystemException)
{
// Your code here.

ACE_Guard<ACE_Mutex> guard(this->lock_);

ACE_DEBUG((LM_INFO, ACE_TEXT("History_i::get_all\n")));

// create a Messages sequence, set its length
::Messages* retval = new ::Messages();
retval->length( this->messages_.size() );

// iterate through the MessageList, copying messages into the return sequence
int i = 0;
for ( MessageList::iterator messageItr = this->messages_.begin();
    messageItr != this->messages_.end();
    ++messageItr ) {

    // because the MessageList contains Message_vars, reference counting
    // upon assignment into the sequence is handled properly for us.
    (*retval)[i++] = *messageItr;
}
return retval;
}

The get_all() operation creates a new Messages sequence, setting its length. It then iterates through the internal STL list of Message_vars, adding each Message to the sequence. Because the STL list stores Message_vars the assignment of each Message from the STL list to the sequence handles memory management properly for us by incrementing the reference count on each returned Message.

The get_latest() operation simply retrieves the last Message added to the list and returns it.

::Message*
History_exec_i::get_latest ()
throw (::CORBA::SystemException)
{
// Your code here.

ACE_Guard<ACE_Mutex> guard(this->lock_);

ACE_DEBUG((LM_INFO, ACE_TEXT("History_i::get_latest\n") ));

// just get the last message from the history. because the MessageList
// contains Message_vars, _var to _var assignment handles the reference
// counting properly for us.
::Message_var retval = this->messages_.back();
return retval._retn();
}

We extract the last Message into a Message_var and return it with the
_retn() operation to handle the reference counting of the Message properly.
We give up ownership of the Message when we return it, but we also want to
keep the Message in the internal list. The reference counting handles that for
us.

The Messenger calls the local add() method to store published Messages.

void History_exec_i::add (::Message* message)
{
    ACE_Guard<ACE_Mutex> guard(lock_);

    // bump up the reference count; we don’t own the parameter.
    // the _var in the STL list takes ownership of the “copy"
    message->_add_ref();
    this->messages_.push_back( message );
}

It increments the reference count of the Message and stores it in the class’s
STL list. If we do not increment the reference count, then the STL list would
attempt to take ownership of a Message that it does not own.

The get_all() and get_latest() operations are exposed to clients
through the History interface. The add() method is not part of the IDL
interface and is visible only through the Messenger implementation.
The Messenger Component Executor

The Messenger component provides the Runnable, Publication, and History facets and publishes Message events. It delegates much of its work to the Runnable, Publication, and History executors.

Recall that the Messenger’s IDL specification is as follows:

```idl
component Messenger {
    attribute string subject;
    provides Runnable control;
    provides Publication content;
    publishes Message message_publisher;
    provides History message_history;
};

home MessengerHome manages Messenger {};
```
3.2 Example - The Messenger Application

The Messenger’s component executor contains a `get_<facet_name>()` operation for each of its three provided facets to expose the facet to the component container. As a general rule, for each IDL3 statement of the form

```idl3
provides <facet_interface> <facet_name>;
```

the CIDL compiler generates a operation of the form

```c
::CCM_<facet_interface>_ptr get_<facet_name>();
```

Thus, the IDL statement

```idl3
provides Publication content;
```

causes the CIDL compiler to generate an operation in the Messenger executor with the signature

```c
::CCM_Publication_ptr get_content();
```

The Messenger’s `MessengerHome` manages the Messenger’s life cycle. The component container creates an instance of the Messenger executor through its `MessengerHome`.

Recall that the Messenger’s CIDL composition is as follows:

```idl3
composition session Messenger_Impl
{
    home executor MessengerHome_Exec
    {
        implements MessengerHome;
        manages Messenger_Exec;
    };
};
```

The CIDL compiler uses both the Messenger’s IDL interface and its CIDL composition to generate an implementation of its executor. It generates a default Messenger executor with empty implementations of the `get_control()`, `get_content()`, and `get_message_history()` facet
accessors. The class diagram illustrates the Messenger executor class hierarchy.

Figure 32-14 Messenger Executor Class Diagram

The IDL compiler does \textit{not} generate a Messenger stub. The Messenger component is not an IDL2 interface. The CIDL compiler generates an abstract executor base class, \texttt{CCM\_Messenger}, just as it did for the Runnable facet. The CIDL compiler also generates a Messenger\_exec class that identifies the Messenger as a \textit{session} component. A session component exports
transient object references and is responsible for managing its own persistent state if it has any.

The CIDL compiler optionally generates an empty executor implementation, \texttt{Messenger\_exec\_i}. The CIDL compiler generates a default constructor, a destructor, and a virtual method for each of the Messenger’s facets.

The executor implementation class also inherits from \texttt{TAO\_Local\_RefCounted\_Object}, marking the \texttt{Messenger\_exec\_i} as a \texttt{CORBA::LocalObject} and allowing the component container to manage the executor’s memory through reference counting.

The CIDL-generated executor implementation is as follows; as always, our changes are in bold. Comments are interspersed through the class definition.

```c
#include "Messenger\_svnt\_h"
#include "Messenger\_exec\_export\_h"
#include "tao/LocalObject\_h"

#include <string>
#include <ace/Task\_h>

namespace Messenger\_Impl
{
    // forward declarations for executor implementations referenced
    // in the Messenger\_exec\_i class definition
    class Runnable\_exec\_i;
    class Publication\_exec\_i;
    class History\_exec\_i;

    class MESSENGER\_EXEC\_Export Messenger\_exec\_i
    : public virtual Messenger\_Exec,
        public virtual TAO\_Local\_RefCounted\_Object,
        public virtual ACE\_Task\_Base
    {
        public:

        Messenger\_exec\_i (void);
        virtual ~Messenger\_exec\_i (void);
    }

    The Messenger executor is an active object, publishing messages in its own thread. It inherits from \texttt{ACE\_Task\_Base} to realize the active object behavior. There will be more on the implications of this later.

    The CIDL compiler generates a default constructor and destructor. There is no reason to change the signatures of these methods.
```
The CIDL compiler generates an empty accessor and modifier for the subject attribute.

```cpp
virtual char* subject ()
  throws (CORBA::SystemException);

virtual void subject (const char* subject)
  throws (CORBA::SystemException);
```

The CIDL compiler generates a get__ operation for each of the Messenger’s three provided facets.

```cpp
virtual ::CCM_Runnable_ptr get_control ()
  throw (CORBA::SystemException);

virtual ::CCM_Publication_ptr get_content ()
  throw (CORBA::SystemException);

virtual ::CCM_History_ptr get_message_history ()
  throw (CORBA::SystemException);
```

The Messenger has three facets: a Runnable facet called control, a Publication facet called content, and a History facet called message_history.

```cpp
  // Operations from Components::SessionComponent
```

The CIDL compiler generates a callback operation to set the component’s session context. It generates a session context class that is specific to the component type. The component container instantiates and sets the component instance’s session context at application startup.

The session context contains methods that enable the component to interact with the other components to which it is connected. Contexts are the glue that plug collaborating components together. As we’ll see later, the Messenger component publishes Message events to interested consumers through its context.

```cpp
virtual void set_session_context (::Components::SessionContext_ptr ctx)
  throw (::CORBA::SystemException,::Components::CCMException);
```
The component container calls `set_session_context()` after it instantiates the component executor instance. The CIDL compiler also generates a protected class member called `context_` to store the context and generates the implementation of the `set_session_context()` operation. No work is required on the part of the component developer.

The CIDL compiler generates three callback operations through which the component container indicates when the component is being activated, passivated, or removed.

```c++
virtual void ccm_activate ()
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ccm_passivate ()
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ccm_remove ()
  throw (::CORBA::SystemException,::Components::CCMException);
```

The component container calls `ccm_activate()` to notify component that it has been activated. The `ccm_activate()` call completes before any other component operations are invoked. The component executor may perform its initialization in `ccm_activate()`. The component developer can assume that the session context has been initialized when the component container calls `ccm_activate()`. The Messenger’s implementation of `ccm_activate()` calls `ACE_Task_Base::activate()` to launch a message-publishing thread.

The component container calls `ccm_passivate()` to notify the component that it has been deactivated. Here, the component executor should release any resources acquired in `ccm_activate()`. The component container then calls `ccm_remove()` when the component executor is about to be destroyed. The component developer can assume that the session context is still available when the component container calls `ccm_passivate()` or `ccm_remove()`.

The `ccm_activate()`, `ccm_passivate()`, and `ccm_remove()` operations are required by the OMG CORBA Component Model specification.

The generated `ciao_preactivate()` and `ciao_postactivate()` operations are CIAO-specific.

```c++
virtual void ciao_preactivate ()
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ciao_postactivate ()
```
throw (::CORBA::SystemException, ::Components::CCMException);

The component container calls ciao_preactivate(), then ccm_activate(), then ciao_postactivate() when activating a component. All three calls happen in the same thread.

If the component container is activating more than one component, it first calls ciao_preactivate() on each component being activated, then calls ccm_activate() on each component being activated, and finally calls ciao_postactivate() on each component being activated.

The svc() method is an implementation detail that is specific to our implementation of the Messenger executor.

    virtual int svc();

It is overridden from the inherited ACE_Task_Base class. Our implementation of ccm_activate() calls ACE_Task_Base::activate() to launch a thread that executes the svc() method. The implementation of the svc() method publishes Message events to interested consumers.

The CIDL compiler automatically generates a context_ class member.

protected:
    CIAO_GLUE::Messenger_Context *context_;

The component container calls set_session_context() to set the context when it initializes the component executor. The Messenger publishes its Message events through the context.

The component developer may add additional class members required to implement the component executor. We add several.

private:
    Runnable_exec_i* control_;
    Publication_exec_i* content_;  
    History_exec_i* history_;  
    std::string subject_;  
    const std::string user_;  
};
The private control, content, and history class members will be initialized by the user’s code to contain pointers to the facet executors for the Runnable, Publication, and History facets of the Messenger component. The user class member is a string that contains a user name that the Messenger embeds into each Message it publishes.

The Messenger executor implementation follows. As always, changes to the CIDL-generated default executor implementation code are in bold. Comments are interspersed with the code.

```cpp
#include "Messenger_exec_i.h"
#include "ciao/CIAO_common.h"

namespace Messenger_Impl
{
  //==================================================================
  // Component Executor Implementation Class:   Messenger_exec_i
  //==================================================================

  Messenger_exec_i::Messenger_exec_i ()
  : subject_("Test Subject"),
    user_("ciao_user")
  {
    // initialize user-defined data members
    this->control_ = new Runnable_exec_i();
    this->history_ = new History_exec_i();
    this->content_ = new Publication_exec_i(
      "Test Subject",
      "The quick brown fox jumped over the lazy dog",
      2
    );
  }
```
The destructor releases the memory for the Messenger’s three facet executors. Because an executor is reference counted, we call \_remove\_ref() to release the memory of the executor rather than use the C++ delete operation.

```cpp
Messenger_exec_i::~Messenger_exec_i (void)
{
    this->control_->\_remove\_ref();
    this->history_->\_remove\_ref();
    this->content_->\_remove\_ref();
}
```

The bulk of the Messenger’s logic is in the \texttt{svc()} method.

```cpp
int Messenger_exec_i::svc() {

    ACE_DEBUG((LM_INFO, ACE_TEXT("svc()\n")));

    while (1)
    {
        ACE_OS::sleep( this->content_->period() );

        // get the run_lock from the Runnable executor; we have an
        // agreement with the Runnable executor that we must wait for
        // the run_lock to be released before we publish.
        ACE_Guard<ACE_Mutex> guard( this->control_->get_run_lock() );

        // create a message to publish
        ::Message_var msg = new ::OBV_Message();
        msg->subject( this->subject() );
        msg->text( this->content_->text() );
        msg->user( CORBA::string_dup( this->user_.c_str() ) );

        // add the message to the message history
        this->history_->add( msg.in() );

        ACE_DEBUG((LM_INFO,
                    ACE_TEXT("Messenger_exec_i::svc: publishing message\n")));

        // publish to all interested consumers
        this->context_->push_message_publisher( msg.in() );

        ACE_DEBUG((LM_INFO,
                    ACE_TEXT("Published Message on subject %s\n  User %s\n  Text %s\n"),
                    msg->subject(),
                    msg->user(),
                    msg->text() ));
    }

    // not reached
}
```
We override the `svc()` method from the inherited `ACE_Task_Base` class. Our implementation of `ccm_activate()` calls the `ACE_Task_Base::activate()` method which launches the `svc()` method in a new thread. This method performs the bulk of the Messenger’s work, looping continuously and publishing messages.

First, the `svc()` method sleeps for the period of time defined by the `period` attribute of the Messenger’s `Publication` executor. Next, the `svc()` method attempts to acquire a lock from its `Runnable` executor. The Messenger executor and the `Runnable` executor have an agreement that the Messenger will not publish messages unless it can acquire the `Runnable` executor’s `ACE_Mutex` lock. This permits the `Runnable` executor to start and stop message publication.

Once the Messenger acquires the `Runnable`’s lock, it publishes a message through its `context_`. The context acts like a proxy representing all interested consumers.

The CIDL compiler generates an empty accessor and modifier for the `subject` attribute. We add an implementation to each.

```c++
char*
Messenger_exec_i::subject ()
  throws (CORBA::SystemException)
{
  return CORBA::string_dup( this->subject_.c_str() );
}

void
Messenger_exec_i::subject (const char* subject)
  throws (CORBA::SystemException)
{
  this->subject_ = CORBA::string_dup( subject );
}
```

The CIDL compiler generates an empty implementation of each of the `get_content()`, `get_control()`, and `get_message_history()` facet accessor operations. We modify each to return the appropriate facet executor, incrementing its reference count before returning it.
The Publication facet controls the Messenger’s message text and publication period. It is important to increment the reference count before returning the facet executor because we give up ownership of the facet executor when we return it. This behavior is consistent with the CORBA’s memory management rules. Notice that we do not need to convert the executor to an object reference; we merely return it an allow the component container to do the heavy lifting.

The implementation of the `get_control()` facet accessor is nearly identical...

```cpp
::CCMRunnable_ptr Messenger_exec_i::get_control ()
    throw (CORBA::SystemException)
{
    // Your code here.

    // bump up ref count because we give up ownership when we return this
    this->control_->_add_ref();
    return this->control_;  
}
```

...as is the implementation of `get_message_history()`.

```cpp
::CCMHistory_ptr Messenger_exec_i::get_message_history ()
    throw (CORBA::SystemException)
{
    // Your code here.

    // bump up ref count because we give up ownership when we return this
    this->history_->_add_ref();
    return this->history_;  
}
```
The CIDL compiler generates a complete implementation of the
set_session_context() operation.

```c++
// Operations from Components::SessionComponent

void
Messenger_exec_i::set_session_context (::Components::SessionContext_ptr ctx)
throw (::CORBA::SystemException,::Components::CCMException)
{
    this->context_ = CIAO_GLUE::Messenger_Context::_narrow (ctx);
    if (this->context_ == 0)
    {
        throw CORBA::INTERNAL ();
    }
}
```

The component container calls this operation immediately after it instantiates
the Messenger executor. The component container calls ccm_activate() to
notify the component that it has been activated.

```c++
void
Messenger_exec_i::ccm_activate ()
throw (::CORBA::SystemException,::Components::CCMException)
{
    // Your code here.
    ACE_DEBUG((LM_INFO, ACE_TEXT("Messenger_exec_i::ccm_activate\n")));
    this->activate();
}
```

The container does not send any requests to the component until
ccm_activate() completes. This is typically where the component executor
initializes itself. The Messenger executor calls the
ACE_Task_Base::activate() method to spawn a thread running the
cvc() method.

The component container calls ccm_passivate() to notify the component
that it has been deactivated and the component container will call
ccm_remove() when the component executor is about to be destroyed. Once
ccm_passivate() is called, the component instance cannot be reactivated.

```c++
void
Messenger_exec_i::ccm_passivate ()
throw (::CORBA::SystemException,::Components::CCMException)
{
Typically, a component executor cleans up after itself in `ccm_passivate()`, where it is guaranteed that the container hasn’t started destroying its other component executors yet. Our executor has nothing to clean up.

The CIDL compiler generates CIAO-specific `ciao_preactivate()` and `ciao_postactivate()` operations.

```cpp
void
Messenger_exec_i::ciao_preactivate ()
  throw (::CORBA::SystemException,:Components::CCMException)
{
  // Your code here.
}

void
Messenger_exec_i::ciao_postactivate ()
  throw (::CORBA::SystemException,:Components::CCMException)
{
  // Your code here.
}
```

The component container calls these CIAO-specific operations before and after it calls `ccm_activate()`, respectively. The CIDL compiler generates these methods with empty implementations. We leave them empty for this example.

### 32.2.4.5 The MessengerHome Executor

The CIDL compiler generates an implementation of the Messenger’s home. Recall that the MessengerHome’s IDL3 interface is as follows:

```idl3
component Messenger { ... };

home MessengerHome manages Messenger {};
```

and the Messenger’s CIDL composition is as follows:
composition session Messenger_Impl
{
    home executor MessengerHome_Exec
    {
        implements MessengerHome;
        manages    Messenger_Exec;
    };
};

The CIDL compiler generates a complete implementation of the MessengerHome executor and a library entry point function. The component container instantiates the MessengerHome through the entry point function when it dynamically loads the Messenger’s library. It generates the MessengerHome executor in the same file as the Messenger component executor.

The MessengerHome is responsible for creating and destroying instances of the Messenger executor. The component container instantiates the Messenger executor through its home when it activates the Messenger. In our example the component developer does not need to modify any of the generated MessengerHome code nor the library entry point function.
The class diagram illustrates the `MessengerHome` executor class hierarchy.

![Class Diagram]

**Figure 32-15 Messenger Home Executor Class Diagram**

Both the class definition and the implementation of the `MessengerHome` and its entry point function are as follows. We make no changes to the generated code; thus, nothing is shown in bold. Comments are interspersed.

```cpp
namespace Messenger_Impl
{
    // Like the Messenger executor, the MessengerHome executor inherits from a generated executor base class.

    // Class definitions and implementations...
```

---

class MESSENGER_EXEC_Export MessengerHome_exec_i
: public virtual MessengerHome_Exec,
   public virtual TAO_Local_RefCounted_Object
{
public:

The CIDL compiler generates a default constructor and destructor.

    MessengerHome_exec_i (void);
    virtual ~MessengerHome_exec_i (void);

The CIDL compiler generates a default create() operation. The component container calls this create() operation to instantiate the Messenger component executor.

    virtual ::Components::EnterpriseComponent_ptr create ()
         throw (::CORBA::SystemException,::Components::CCMException);

};

Finally, the CIDL compiler generates a library entry point function. The component container’s underlying ACE Service Configurator calls this entry point function to instantiate the MessengerHome executor when it dynamically loads the component’s library. The entry point function must have “C” linkage to prevent C++ name mangling. The name of the function is CIAO-specific, but every CCM implementation generates an entry point function with this signature.

extern "C" MESSENGER_EXEC_Export ::Components::HomeExecutorBase_ptr
    createMessengerHome_Impl (void);
}

As you can see, we also make no changes to the generated MessengerHome implementation, which follows:

//==================================================================
// Home Executor Implementation Class:   MessengerHome_exec_i
//==================================================================

The CIDL compiler generates a default constructor and an empty destructor. The component developer may modify these. However, if the developer modifies the signature of the constructor, the developer must modify the
implementation of the component home’s library entry point function to pass the appropriate constructor arguments. The library entry point function will be shown in a few paragraphs.

MessengerHome_exec_i::MessengerHome_exec_i (void)
{
}

MessengerHome_exec_i::~MessengerHome_exec_i (void)
{
}

The Messenger’s home has one operation, create(). The CIDL-generated implementation of create() simply invokes the Messenger executor’s default constructor.

::Components::EnterpriseComponent_ptr
MessengerHome_exec_i::create ()
throw (::CORBA::SystemException, ::Components::CCMException)
{
::Components::EnterpriseComponent_ptr retval =
::Components::EnterpriseComponent::_nil ();

ACE_NEW_THROW_EX (retval,
    Messenger_exec_i,
    CORBA::NO_MEMORY ());
ACE_CHECK_RETURN (::Components::EnterpriseComponent::_nil ());

return retval;
}

Finally, the CIDL compiler generates a library entry point function for the Messenger. This function simply creates an instance of the Messenger’s home executor. The component container calls this function to create a MessengerHome when it loads the Messenger’s dynamic library.

extern "C" MESSENGER_EXEC_Export ::Components::HomeExecutorBase_ptr
createMessengerHome_Impl (void)
{
::Components::HomeExecutorBase_ptr retval =
::Components::HomeExecutorBase::_nil ();

ACE_NEW_RETURN (retval,
    MessengerHome_exec_i,
    ::Components::HomeExecutorBase::_nil ());
32.2.4.6 The Receiver Component Executor

The Receiver component connects to the Messenger component in two ways. First, its `message_consumer` port connects to the Messenger’s `message_publisher` port. Second, its `message_history` receptacle connects to the Messenger’s `message_history` facet.

Recall that the Receiver’s IDL specification is as follows:

```idl
component Receiver {
    consumes Message message_consumer;
    uses History message_history;
};

home ReceiverHome manages Receiver {};
```
The Receiver component subscribes to Message events and uses a History facet through which it retrieves a history of messages published. Both of these facets happen to be provided by the Messenger component, but the Receiver does not know that and does not need to know that. In fact, the Message events could be published by several different suppliers without the Receiver being aware of it.

The Receiver’s ReceiverHome manages the Receiver’s life cycle. The component container creates an instance of the Receiver executor through its ReceiverHome.

Recall that the Receiver’s composition is as follows:

```
composition session Receiver_Impl
{
    home executor ReceiverHome_Exec
    {
        implements ReceiverHome;
        manages Receiver_Exec;
    }
};
```

There are many similarities between the Receiver’s executor and the Messenger’s executor. As with the Messenger, the CIDL compiler uses both the Receiver’s IDL interface and its CIDL composition to generate its executor. The CIDL compiler generates a class definition for the Receiver executor, another for the ReceiverHome, and a library entry point function. The CIDL compiler generates the Receiver’s executor classes into the Receiver_Impl C++ namespace as specified by the Receiver’s CIDL composition.

The most noticeable difference between the Messenger and the Receiver executors is the Receiver’s message consumption callback operation. As a general rule, an IDL3 statement of the form

```
consumes <event_type> <facet_name>;
```

causes the CIDL compiler to generate a callback operation in the component’s executor of the form:

```
virtual void push_<facet_name>({ <event_type>* ev });
```

In our example, the IDL statement
consumes Message message_consumer;

causes the CIDL compiler to generate a callback operation with the signature

virtual void push_message_consumer( Message* ev );

The component container calls this operation when a connected Message supplier (in our case, the Messenger) publishes a Message event. The component container connects the suppliers and consumers dynamically at deployment time.

The CIDL-generated Receiver and ReceiverHome executor implementation class definitions are as follows. We make no changes to the generated Receiver executor class definition. Comments are interspersed.

```
#include "Receiver_svnt.h"
#include "Receiver_exec_export.h"
#include "tao/LocalObject.h"

namespace Receiver_Impl
{

    The Receiver’s executor implements the abstract executor base class Receiver_exec.

    class RECEIVER_EXEC_Export Receiver_exec_i
    : public virtual Receiver_Exec,
     public virtual TAO_Local_RefCounted_Object
    {
    
        The CIDL compiler generates a default constructor and a destructor.

        public:
            Receiver_exec_i (void);
            virtual ~Receiver_exec_i (void);

        The component container calls push_message_consumer() on the Receiver when the Messenger publishes a Message.

            virtual void push_message_consumer (::Message *ev)
            throw (CORBA::SystemException);

```
Like the Messenger component, the Receiver component has a CIDL-generated set_session_context() callback operation. Again, the component container calls this operation after it instantiates the Receiver executor.

```cpp
virtual void set_session_context (::Components::SessionContext_ptr ctx) throw (::CORBA::SystemException,::Components::CCMException);
```

The CIDL compiler also generates standard ccm_activate(), ccm_passivate(), and ccm_remove() operations and CIAO-specific ciao_preactivate() and ciao_postactivate() operations.

```cpp
virtual void ccm_activate () throw (::CORBA::SystemException,::Components::CCMException);
virtual void ccm_passivate () throw (::CORBA::SystemException,::Components::CCMException);
virtual void ccm_remove () throw (::CORBA::SystemException,::Components::CCMException);
virtual void ciao_preactivate () throw (::CORBA::SystemException,::Components::CCMException);
virtual void ciao_postactivate () throw (::CORBA::SystemException,::Components::CCMException);
```

Finally, the CIDL compiler generates type-specific context member for the Receiver component.

```cpp
protected:
  CIAO_GLUE::Receiver_Context *context_;
};
```

Our Receiver executor has no state nor private methods, so we make no changes to the CIDL-generated Receiver executor class definition.

The CIDL compiler also generates an implementation of the Receiver executor. The bulk of the work in implementing the Receiver executor is in the push_message_consumer() operation. The push_message_consumer() implementation uses the Receiver’s message_history facet to get a list of all Message events published. The Receiver accesses the message_history facet through its session context.
As a general rule, an IDL3 statement of the form

uses <facet_interface> <facet_name>;

is mapped to a C++ function in the component’s session context with the signature

::<facet_interface>_ptr get_connection_<facet_name>();

Thus, the Receiver’s IDL3 statement

uses History message_history;

is mapped to a C++ function in the Receiver’s context with the signature

::History_ptr get_connection_message_history();

Note The Receiver specifies the services that it requires through the uses statement. This syntax enables the dynamic connection of service providers and service users at run time.

The Receiver executor implementation follows. Changes to CIDL-generated executor implementation code are noted in bold:

```
#include "Receiver_exec_i.h"
#include "ciao/CIAO_common.h"

namespace Receiver_Impl
{
    // Component Executor Implementation Class:  Receiver_exec_i
    //================================================================================

    Receiver_exec_i::Receiver_exec_i (void)
    {
    }

    Receiver_exec_i::~Receiver_exec_i (void)
    {
    }
```
The component container invokes the push_message_consumer() operation a connected supplier publishes a Message event.

```c++
void Receiver_exec_i::push_message_consumer (::Message * ev)
throw (CORBA::SystemException)
{
  // Your code here.

  CORBA::String_var subject = ev->subject();
  CORBA::String_var user = ev->user();
  CORBA::String_var text = ev->text();

  ACE_DEBUG(
    (LM_INFO,
      ACE_TEXT("Received Message:\n Subject: %s\n User: %s\n Text: %s\n"),
      subject.in(),
      user.in(),
      text.in() ));

  // Use the history to (inefficiently) get the total number of messages
  // published on this item so far
  ::History_var history = this->context_->get_connection_message_history();
  ::Messages_var messages = history->get_all();
  ACE_DEBUG((LM_INFO,
      ACE_TEXT(" Subject "%s" has published %d messages so far\n"),
      subject.in(),
      messages->length() ));
}
```

The implementation of push_message_consumer() prints out the message’s subject, user, and text. Then, it gets the component’s message_history facet, gets a list of all Message events published, and prints out the number of messages published so far.

As with the Messenger component, the CIDL compiler generates a complete implementation of the set_session_context() method.

```c++
// Operations from Components::SessionComponent

void Receiver_exec_i::set_session_context (::Components::SessionContext_ptr ctx)
throw (::CORBA::SystemException,::Components::CCMException)
{
  this->context_ = CIAO_GLUE::Receiver_Context::_narrow (ctx)

  if (this->context_ == 0)
    {
```
The component container calls the `set_session_context()` operation to set the Receiver’s context immediately after it instantiates the Receiver executor. The Receiver uses this context to access its connected `message_history` facet. We do not make any changes to this method.

The CIDL compiler generates empty implementations of `ccm_activate()`, `ccm_passivate()`, and `ccm_remove()` as well as empty implementations of CIAO-specific methods `ciao_preactivate()` and `ciao_postactivate()`. We do not modify any of these methods.

```c++
void Receiver_exec_i::ccm_activate ()
    throw (::CORBA::SystemException, ::Components::CCMException)
{
    // Your code here.
}

debug Receiver_exec_i::ccm_passivate ()
    throw (::CORBA::SystemException, ::Components::CCMException)
{
    // Your code here.
}

debug Receiver_exec_i::ccm_remove ()
    throw (::CORBA::SystemException, ::Components::CCMException)
{
    // Your code here.
}

debug Receiver_exec_i::ciao_preactivate ()
    throw (::CORBA::SystemException, ::Components::CCMException)
{
    // Your code here.
}

debug Receiver_exec_i::ciao_postactivate ()
    throw (::CORBA::SystemException, ::Components::CCMException)
{
    // Your code here.
}
```
32.2.4.7 The ReceiverHome Executor

Recall that the ReceiverHome’s IDL3 interface is as follows:

```cpp
cOMPONENT Receiver { ... };

home ReceiverHome manages Receiver {};
```

and the Receiver’s CIDL composition is as follows:

```cpp
COMPOSITION session Receiver_Impl
{
    home executor ReceiverHome_Exec
    {
        implements ReceiverHome;
        manages Receiver_Exec;
    }
};
```

The ReceiverHome class definition and implementation are nearly identical to the MessengerHome class definition and implementation. We make no changes to the CIDL-generated code.

```cpp
class RECEIVER_EXEC_Export ReceiverHome_exec_i
  : public virtual ReceiverHome_Exec,
    public virtual TAO_Local_RefCounted_Object
{
  public:
    ReceiverHome_exec_i (void);
    virtual ~ReceiverHome_exec_i (void);

    virtual ::Components::EnterpriseComponent_ptr create ()
      throw (::CORBA::SystemException, ::Components::CCMException);
  
  extern "C" RECEIVER_EXEC_Export ::Components::HomeExecutorBase_ptr
    createReceiverHome_Impl (void);
}
```

Finally, the CIDL compiler generates implementations of the ReceiverHome executor and the Receiver’s library entry point function. We do not modify these implementations.

```cpp
//====================================================================
```
32.2 Example - The Messenger Application

32.2.4.8 The Administrator Component Executor

The Administrator component starts and stops the Messenger’s message publication and controls the published text and the publication period. The Administrator component demonstrates use of the uses multiple mechanism of connecting a component receptacle to multiple facets simultaneously.
Recall that the Administrator’s IDL3 interface is as follows:

```idl3
component Administrator {
    uses multiple Runnable runnables;
    uses multiple Publication content;
};
```

```idl3
home AdministratorHome manages Administrator();
```

At deployment time, we connect the Administrator’s runnables and content receptacles to the Messenger’s Runnable and Publication facets. The Administrator controls the starting and stopping of message publication through the Runnable facet and controls the message text published and the publication period through the Publication facet.

The uses multiple modifier on the runnables and content receptacles permits the Administrator to connect an unlimited number of Runnable and Publication facets. This type of receptacle is a *multiplex receptacle*. Our example uses only one of each; however, an application deployer may add...
additional Messengers into the application through XML-based configuration at deployment time. More information on deployment is available in 32.2.5. Recall that the Administrator’s composition, describing the implementation of the Administrator component, is as follows:

```
composition session Administrator_Impl
{
    home executor AdministratorHome_Exec
    {
        implements AdministratorHome;
        manages Administrator_Exec;
    };
}
```

The CIDL compiler generates the Administrator’s executor implementation code into an `Administrator_Impl` namespace. It generates an `Administrator_exec` executor base class for the Administrator and an `AdministratorHome_exec` executor base class for the Administrator’s home. In addition, the CIDL compiler optionally generates default implementations of the executors. These are analogous to the CIDL-generated classes for the Messenger and Receiver components.

It is not surprising that there are many similarities between the Administrator executor and the Messenger and Receiver executors. The most noticeable change we make to the CIDL-generated executor implementation is to add inheritance from the `ACE_Task_Base` class and override the `ACE_Task_Base::svc()` method. The Administrator’s implementation launches a thread that displays a simple text menu to start, stop, and otherwise control the Messenger’s message publication. A more realistic application might use a GUI of some kind. The purpose of this example implementation is to demonstrate that a user can manually interact with a deployed component.

The Administrator’s IDL3 interface introduces the `uses multiple receptacle`, or `multiplex receptacle`. A multiplex receptacle can connect to an unlimited number of type-compatible facets. The mapping from IDL to C++ is different for a multiplex receptacle than for a `simplex receptacle`, which connects to exactly one facet.

You might remember that for a receptacle of the form

```
uses <facet_interface> <facet_name>;
```
the CIDL compiler generates a C++ function in the component’s context with the signature

::<facet_interface>_ptr get_connection_<facet_name>();

For example,

uses History message_history;

is mapped to a C++ function in the component’s context with the signature

::History_ptr get_connection_message_history();

This type of receptacle is called a simplex receptacle because it connects to exactly one facet. However, a *multiplex* receptacle of the form

uses multiple <facet_interface> <facet_name>;

causes the IDL and CIDL compilers to generate the following:

- A C++ struct called `<facet_name>Connection`:

```
struct <facet_name>Connection
{
    typedef <facet_name>Connection_var _var_type;
    <facet_type>_var objref;
    Components::Cookie_var ck;
};
```

- A typedef for a sequence of `<facet_name>Connection` elements called `<facet_name>Connections`. *We illustrate with the equivalent IDL2:*

```
typedef sequence< <facet_name>Connection > <facet_name>Connections;
```

- A facet accessor C++ function through which the component retrieves a sequence of connected facets:

```
::<component_name>::<facet_name>Connections* get_connections_<facet_name>();
```

In our example, the multiplex Runnable receptacle

uses multiple Runnable runnables;
32.2 Example - The Messenger Application

causes the IDL and CIDL compilers to generate the following:

- A C++ struct called runnablesConnection:

```cpp
struct runnablesConnection
{
    typedef runnablesConnection_var _var_type;
    Runnable_var objref;
    Components::Cookie_var ck;
};
```

- A C++ sequence type of runnablesConnection elements called runnablesConnections. Again, we illustrate with the equivalent IDL2:

```cpp
typedef sequence< runnablesConnection > runnablesConnections;
```

- A facet accessor C++ function through which the component retrieves a sequence of connected Runnable facets:

```cpp
::Administrator::runnablesConnections* get_connections_runnables();
```

The startPublishing(), stopPublishing(), changePublicationText(), and changePublicationPeriod() helper methods illustrate the usage of the multiplex receptacle.

The Administrator executor follows. Changes to CIDL-generated code are in **bold**.

```cpp
#include "Administrator_svnt.h"
#include "Administrator_exec_export.h"
#include "tao/LocalObject.h"
#include <ace/Task.h>

namespace Administrator_Impl
{
    class ADMINISTRATOR_EXEC_Export Administrator_exec_i
        : public virtual Administrator_Exec,
          public virtual ACE_Task_Base,
          public virtual TAO_Local_RefCounted_Object
    
    public:
```
The remainder of the executor class definition is the analogous to the Messenger’s and Receiver’s...

```cpp
Administrator_exec_i (void);
virtual ~Administrator_exec_i (void);

virtual void set_session_context (::Components::SessionContext_ptr ctx)
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ccm_activate ()
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ccm_passivate ()
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ccm_remove ()
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ciao_preactivate ()
  throw (::CORBA::SystemException,::Components::CCMException);

virtual void ciao_postactivate ()
  throw (::CORBA::SystemException,::Components::CCMException));

protected:
  CIAO_GLUE::Administrator_Context *context_;

...except that we override the ACE_Task_Base::svc() method and add several private helper methods to control the connected Runnable and Publication facets.

public:
  // Overridden from ACE_Task_Base
  int svc();

private:
  void startPublishing();
  void stopPublishing();
  void changePublicationPeriod();
  void changePublicationText();
};
```

The Administrator’s executor implementation follows, with changes to CIDL-generated code in **bold**:

```
#include "Administrator_exec_i.h"
```
32.2 Example - The Messenger Application

```cpp
#include "ciao/CIAO_common.h"

#include <iostream>
#include <string>

namespace Administrator_Impl
{
    // Component Executor Implementation Class: Administrator_exec_i
    // Functions not modified

    Administrator_exec_i::Administrator_exec_i (void)
    {
    }
    Administrator_exec_i::~Administrator_exec_i (void)
    {
    }

    void Administrator_exec_i::set_session_context (::Components::SessionContext_ptr ctx)
    throw (::CORBA::SystemException,::Components::CCMException)
    {
        this->context_ = CIAO_GLUE::Administrator_Context::_narrow (ctx);
        if (this->context_ == 0)
        {
            throw CORBA::INTERNAL ();
        }
    }

    // In the ccm_activate() implementation, the executor calls
    // ACE_Task_Base::activate() to launch its svc() method in a thread.

    void Administrator_exec_i::ccm_activate ()
    throw (::CORBA::SystemException,::Components::CCMException)
    {
        // Your code here.

        // Activate the Task, triggering its svc() method
        this->activate();
    }
```
We have no need to modify any of the remaining generated operations.

```cpp
void Administrator_exec_i::ccm_passivate ()
  throw (::CORBA::SystemException,::Components::CCMException)
{
  // Your code here.
}

void Administrator_exec_i::ccm_remove ()
  throw (::CORBA::SystemException,::Components::CCMException)
{
  // Your code here.
}

void Administrator_exec_i::ciao_preactivate ()
  throw (::CORBA::SystemException,::Components::CCMException)
{
  // Your code here.
}

void Administrator_exec_i::ciao_postactivate ()
  throw (::CORBA::SystemException,::Components::CCMException)
{
  // Your code here.
}
```

The Administrator’s implementation of the `ccm_activate()` launches a thread and calls `svc()`. The `svc()` method creates a small text menu through which a user can start and stop message publication, change the message text, and change the publication period for every Messenger attached to the Administrator. Helper methods implement those behaviors. Nothing in the implementation of the `svc()` method involves the CORBA Component Model. Code relevant to CCM is in the helper methods, described later.

```cpp
int Administrator_exec_i::svc()
{
  enum SelectionType { START=1, STOP, CHANGE_PERIOD, CHANGE_TEXT };

  while ( 1 ) {
    std::cout << "\nWhat do you want to do to the Messenger(s)?" << std::endl;
    std::cout << START << "  Start" << std::endl;
    std::cout << STOP  << "  Stop"   << std::endl;
    ..
  }
}
```
32.2 Example - The Messenger Application

```cpp
std::cout << CHANGE_PERIOD << ". Change Publication Period" << std::endl;
std::cout << CHANGE_TEXT << ". Change Publication Text" << std::endl;

char selection_text[10];
std::cout << "Please enter a selection: ";
std::cin.getline( selection_text, sizeof(selection_text) );
int selection = ACE_OS::atoi(selection_text);

switch (selection) {
  case START:
    startPublishing();
    break;
  case STOP:
    stopPublishing();
    break;
  case CHANGE_PERIOD:
    changePublicationPeriod();
    break;
  case CHANGE_TEXT:
    changePublicationText();
    break;
  default:
    std::cout << "Please enter a valid option" << std::endl;
    break;
}
return 0;
```

The following four developer-written methods are helper methods invoked by `svc()` in response to user interaction. The first method, `startPublishing()`, retrieves the `Runnable` facets connected to the `Administrator`'s `runnables` receptacle and invokes `start()` on each one.

```cpp
void Administrator_exec_i::startPublishing()
{
  // Get the attached Runnable facet(s)
  ::Administrator::runnablesConnections_var connections =
    this->context_->get_connections_runnables();

  std::cout << "Starting Publication" << std::endl;
  for ( unsigned int i = 0; i < connections->length(); ++i ) {
    Runnable_var runnable = (*connections)[i].objref;
    runnable->start();
  }
}
```

In `startPublishing()`, we use the `Administrator` context's `get_connections_runnables()` method to get a list of the `Runnable`
facets connected to the Administrator’s runnables receptacle. That method returns a sequence of runnablesConnection structs. One of the members of the runnablesConnection struct is a Runnable object reference called objref. We pull the object reference out of the struct and call start() on it to start message publication.

The stopPublishing() method is almost identical to the startPublishing() method...

```cpp
void Administrator_exec_i::stopPublishing()
{
    // Get the attached Runnable facet(s)
    ::Administrator::runnablesConnections_var connections =
        this->context_->get_connections_runnables();

    std::cout << "Stopping Publication" << std::endl;
    for ( unsigned int i = 0; i < connections->length(); ++i ) {
        Runnable_var runnable = (*connections)[i].objref;
        runnable->stop();
    }
}
```

..., except it calls stop() on each connected Runnable facet instead of start().

The changePublicationPeriod() and changePublicationText() methods are similar to startPublishing() and stopPublishing(). The main difference is that they operate on the Publication receptacle rather than the Runnables receptacle.

```cpp
void Administrator_exec_i::changePublicationPeriod()
{
    // Get the attached Publication facet(s)
    ::Administrator::contentConnections_var contents =
        this->context_->get_connections_content();

    char period[10];
    std::cout << "Please enter a new period in seconds: ";
    std::cin.getline( period, sizeof( period ) );
    for ( unsigned int i = 0; i < contents->length(); ++i ) {
        Publication_var publication = (*contents)[i].objref;
        publication->period( ACE_OS::atoi(period) );
    }
}
```
In `changePublicationPeriod()`, we use the Administrator context’s `get_connections_content()` operation to get a list of the Publication facets connected to the Administrator’s content receptacle. That method returns a sequence of `contentConnection` structs. One of the members of the `contentConnection` struct is a Publication object reference called `objref`. We pull the object reference out of the struct and call `period()` on it to change the message publication period.

The `changePublicationText()` method is nearly identical to `changePublicationPeriod()`...

```cpp
void Administrator_exec_i::changePublicationText()
{
    // Get the attached Publication facet(s)
    ::Administrator::contentConnections_var contents = this->context_->get_connections_content();

    char buffer[1024];
    std::cout << "Please enter new text: ";
    std::cin.getline( buffer, sizeof(buffer) );
    for ( unsigned int i = 0; i < contents->length(); ++i ) {
        Publication_var publication = (*contents)[i].objref;
        publication->text( buffer );
    }
}
```

... except it calls the `text()` method on each Publication object reference to change the published text.

### 32.2.4.9 The AdministratorHome Executor

Recall that the AdministratorHome’s IDL3 interface is as follows:

```idl
component Administrator { ... };

home AdministratorHome manages Administrator {};
```

and the Administrator’s CIDL composition is as follows:

```idl
composition session Administrator_Impl
{
    home executor AdministratorHome_Exec
    {
        implements AdministratorHome_Exec;
        manages Administrator_Impl;
    }
}
```
The AdministratorHome class definition and implementation are nearly identical to the MessengerHome and ReceiverHome class definition and implementation. As with those, the CIDL compiler can optionally generate the full class definition, implementation, and library entry point function.

```cpp
class ADMINISTRATOR_EXEC_Export AdministratorHome_exec_i
 : public virtual AdministratorHome_Exec,
  public virtual TAO_Local_RefCounted_Object
{
public:
    AdministratorHome_exec_i (void);
    virtual ~AdministratorHome_exec_i (void);

    virtual ::Components::EnterpriseComponent_ptr create ()
      throw (::CORBA::SystemException, ::Components::CCMException);
};

extern "C" ADMINISTRATOR_EXEC_Export ::Components::HomeExecutorBase_ptr
    createAdministratorHome_Impl (void);
```

The implementation of the class and the library entry point function follows.

```cpp
//==================================================================
// Home Executor Implementation Class:   AdministratorHome_exec_i
//==================================================================
AdministratorHome_exec_i::AdministratorHome_exec_i (void)
{
}

AdministratorHome_exec_i::~AdministratorHome_exec_i (void)
{
}

::Components::EnterpriseComponent_ptr
AdministratorHome_exec_i::create ()
  throw (::CORBA::SystemException, ::Components::CCMException)
{
    ::Components::EnterpriseComponent_ptr retval =
    ::Components::EnterpriseComponent::nil () ;

    ACE_NEW_THROW_EX (retval,
        Administrator_exec_i,
        CORBA::NO_MEMORY ());
```

1284

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. 
Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
32.2 Example - The Messenger Application

ACE_CHECK_RETURN (::Components::EnterpriseComponent::_nil ());

return retval;
}

extern "C" ADMINISTRATOR_EXEC_Export ::Components::HomeExecutorBase_ptr
createAdministratorHome_Impl (void)
{
::Components::HomeExecutorBase_ptr retval =
::Components::HomeExecutorBase::_nil ();

ACE_NEW_RETURN (retval,
    AdministratorHome_exec_i,
    ::Components::HomeExecutorBase::_nil ());

return retval;
}

32.2.4.10 Summary of the Code

The code for our Messenger example is complete. We have implemented executors for the Messenger, Receiver, and Administrator components. We also implemented executors for the Runnable, Publication, and History facets of the Messenger component.

We have seen how the component container injects a context into each component executor to facilitate connections between facets and receptacles and between publishers and consumers.

Consider what we have not done. We have not written a main(). We have not interacted with the Portable Object Adapter, nor have we been exposed to any classes with the prefix POA. We have not written any code that attempts to find another component or object; instead, those connections are provided through each component’s context.

In the following sections we deploy the Messenger example through CIAO’s implementation of the CCM Deployment and Configuration specification.

32.2.5 Deploying the Messenger Application

As the previous sections have demonstrated, a CCM-based application consists of small, self-contained, reusable software components defined using IDL3. A component defines its interactions with other components via ports indicating provided and required interfaces and messages published and
consumed. The deployment activity separates business application logic from process and interaction details.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet

**Describe the application’s deployment**

- Describe each component’s libraries and ports
- Connect component instances through their ports
- Deploy each component into a component container
- Build the application
- Run the application

**Figure 32-18 Road Map**

An assembly is a set of interconnected component instances. Each assembly is itself a component; an assembly may be deployable as a full application, or may represent a higher-level component for use in a larger application. An assembly is a realization of the composite pattern; an assembly may be composed of other assemblies and may be a part of other assemblies.

A component deployer deploys an assembly into one or more application servers that might be distributed across a network. XML descriptor files describe how the components are plugged together into an assembly and how each component of the assembly is mapped to an application server process. Thus, component assembly and deployment is completely independent of component implementation, achieving a separation of concerns.

Each component may be platform-specific, but an assembly may be heterogeneous. A component implementation neither knows nor cares which assemblies it may be a part of nor knows to which other components it may connect.
Our deployment of the Messenger application is described by the following UML deployment diagram.

![UML Deployment Diagram](image)

**Figure 32-19 Deployment of the Messenger Application**

The deployment consists of one Messenger instance, two Receiver instances, and one Administrator instance. We deploy each instance in its own component server, although it is not necessary to do that. In our deployment, the four component servers execute on the same host, although we could distribute them across the network with no code changes and minimal configuration changes.

The “Deployment and Configuration of Component-based Distributed Applications” specification (OMG Document ptc/03-07-08) prescribes how components are instantiated, connected, and assigned to processes in a
distributed software system. The following sections illustrate the salient points of the D&C specification through the deployment of the Messenger application.

The example’s XML descriptor files are in the $TAO_ROOT/DevGuideExamples/CIAO/Messenger/descriptors directory.

### 32.2.5.1 Deployment and Configuration Specification

The OMG’s specification for the “Deployment and Configuration of Component-based Distributed Applications” is a deployment and configuration specification that is independent of the CORBA Component Model. It describes a general-purpose deployment and configuration framework for use by any component-based application. The specification defines IDL interfaces and XML descriptor file formats for configuring individual components and component assemblies for deployment.

This Deployment and Configuration specification supersedes the “Packaging and Deployment” chapter of the OMG CORBA Component Model (CCM) (OMG Document formal/02-06-65) specification.

CIAO’s realization of the Deployment and Configuration specification is called DAnCE (Deployment And Configuration Engine). DAnCE supersedes CIAO’s implementation of the CCM specification’s “Packaging and Deployment” chapter. We deploy the Messenger application using DAnCE in this section.

### 32.2.5.2 Deployment Descriptors

The Deployment and Configuration specification presents a set of XML descriptors for describing deployment aspects of a software system. Each component has a set of descriptors to define its libraries, exposed ports, and implementation. Each application consists of one or more assemblies that describe the application’s packaging and deployment onto component servers.

A CCM application deployer writes many deployment descriptor files to describe the application’s deployment. These files are written by hand. Usually, an application deployer copies and edits an existing set of XML deployment descriptors to describe a new application’s deployment. In the future, we might expect tools such as Vanderbilt’s CoSMIC or Kansas State’s
Cadena to generate the bulk of our CIAO application’s deployment descriptors.

A component such as the Messenger component describes its deployment using the following descriptors:

- An Implementation Artifact Descriptor (.iad) file for each of the Messenger’s libraries.
- A CORBA Component Descriptor (.ccd) file defining the Messenger’s exposed ports and attributes.
- A Component Implementation Descriptor (.cid) file describing the component’s implementation in terms of its Implementation Artifact Descriptors and its CORBA Component Descriptor.
- A Component Package Descriptor (.cpd) file describing one or more implementations of the component.

An assembly is a composite component, consisting of interconnected subcomponents. We assemble a Messenger component instance, two Receiver component instances, and an Administrator component instance into a deployable assembly. The assembly describes its deployment using the following descriptors:

- Another CORBA Component Descriptor (.ccd) file defining the assembly’s exposed ports and attributes, if any.
- Another Component Implementation Descriptor (.cid) file describing the assembly’s implementation in terms of its subcomponents and the connections between their ports.
- Another Component Package Descriptor (.cpd) file describing one or more implementations of the assembly.

An application consists of one or more assemblies. The application describes its deployment using the following descriptors:

- A Package Configuration Descriptor (.pcd) that describes a deployable component package.
- A Top-level Package Descriptor (package.tpd) that contains one or more Package Configuration Descriptors.
• A *Component Deployment Plan* (.cdp) that maps each component instance in the assembly’s Component Implementation Description to a logical node.

• A *Component Domain Descriptor* (.cdd) that describes available nodes, interconnects, and bridges.

• A *Node Map* that maps each logical node to a physical component server process.

The deployment engineer may choose to combine several of these descriptor files into one. In fact, we could describe the Messenger application’s deployment with one rather large XML descriptor. However, we will keep the descriptor files separate for maximum flexibility.

The following sections outline the deployment of the individual components, the Messenger assembly, and the full application.

### 32.2.5.3 Deployment.xsd and XML.xsd Files

The deployment descriptors are described by two XML Schema Definition (XSD) files: Deployment.xsd and XMI.xsd. The directory containing the Messenger application’s deployment descriptors must contain a copy of each of these files. Each file can be copied from DAnCE’s root directory, $CIAO_ROOT/DAnCE or %CIAO_ROOT%\DAnCE.
32.2.5.4 Deploying the Messenger Component

The Messenger component’s deployment descriptors package its libraries and its exposed ports into one deployable package.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - **Describe each component’s libraries and ports**
    - Connect component instances through their ports
    - Deploy each component into a component container
- Build the application
- Run the application

Figure 32-20 Road Map

Six XML descriptors describe the Messenger component’s deployment: three descriptors describe each of the Messenger’s three libraries; one descriptor describes its exposed ports; one descriptor combines the library descriptors and the port descriptor into an implementation; and one descriptor packages the Messenger into a deployable package.
The drawing illustrates the relationships between the six Messenger deployment descriptors.

**Figure 32-21 Messenger Component Deployment Descriptors**

The Messenger’s implementation is linked into three dynamic libraries: Messenger_stub, containing the Messenger’s IDL-generated stub code; Messenger_svnt, containing the Messenger’s IDL- and CIDL-generated skeleton and servant code; and Messenger_exec, containing the Messenger’s home and component executors. We create an Implementation Artifact Descriptor for each of these libraries.

A CORBA Component Descriptor describes the Messenger’s public interface. It contains information about each of the component’s ports, including the
port name, the type of port (Facet, EventPublisher, EventConsumer, etc.) and the port’s supported IDL interfaces.

The Component Implementation Descriptor describes the Messenger component’s monolithic implementation. A monolithic component implementation consists of one or more implementation artifacts. In a C++ application, an implementation artifact is a dynamic library.

At the top level, a Component Package Descriptor may describe several alternate implementations of a component, permitting the component server to choose the correct implementation at run-time based on platform and QoS requirements. Our example provides one implementation of the Messenger component.

**Messenger Component - Implementation Artifact Descriptors (.iad)**

An implementation artifact is a library, a jar file, or some other artifact containing a component’s executable code. We create an Implementation Artifact Descriptor for each of our three Messenger libraries. We also create a separate Implementation Artifact Descriptor for the ACE, TAO, and CIAO libraries.

The Implementation Artifact Descriptor for the ACE, TAO, and CIAO libraries is as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
<label>ACE/TAO/CIAO Libraries</label>
<location>$ACE_ROOT/lib/ACE</location>
<location>$ACE_ROOT/lib/TAO</location>
<location>$ACE_ROOT/lib/CIAO_Client</location>
</Deployment:ImplementationArtifactDescription>
```

The optional `<label>` element contains a human-readable description of the implementation artifact. It may be used by a tool for display purposes. The mandatory `<location>` elements reference the ACE, TAO and CIAO_Client libraries that the Messenger depends upon. File extensions for the libraries are not necessary, or even desired. Multiple alternate location for the same entity can be provided. The underlying implementation uses the operating system’s
dynamic library capabilities, meaning that it can use the contents of the **PATH** and/or the **LD_LIBRARY_PATH** to find the dynamic libraries.

---

**Note**  
Notice the simplicity of the specified library names. The **ACE** library is specified merely as **ACE**, not as **ACE.dll**, **ACEd.dll**, **libACE.so**, etc. The simplified name enables the component developer to describe an implementation artifact in a platform-independent manner. This behavior is specific to CIAO.

---

An Implementation Artifact Descriptor describes each of the three Messenger libraries. The **Messenger_stub** library is described as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
 <label>Messenger Stub Artifact</label>
 <location>Messenger_stub</location>
 <dependsOn>
   <name>ACE/TAO/CIAO</name>
   <referencedArtifact href="Libraries.iad"/>
 </dependsOn>
</Deployment:ImplementationArtifactDescription>
```

The optional `<label>` element may be used by a tool for display purposes. The `<location>` element contains the simplified name of the library. Since we have not provided a path, the library must be in the application’s **PATH** (Windows) or **LD_LIBRARY_PATH** (UNIX). Optionally, we could provide a path, as we did for the ACE, TAO, and CIAO libraries.

Each `<dependsOn>` element contains references to dependent implementation artifacts. We depend on the ACE/TAO/CIAO libraries, so our `<dependsOn>` entry references the **Libraries.iad** file containing references to the ACE/TAO/CIAO libraries. The mandatory `<name>` sub-element may be used by a tool for display purposes.

The descriptor also recognizes one or more optional `<infoProperty>` elements that provide non-functional information that might be displayed by a tool. For example:
The Messenger has two more Implementation Artifact Descriptors, one for its 
Messenger_svnt library and one for its Messenger_exec library. The 
Messenger_svnt.iad files is as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
 <label>Messenger Servant Artifact</label>
 <location>Messenger_svnt</location>
 <dependsOn>
  <name>ACE/TAO/CIAO</name>
  <referencedArtifact href="Libraries.iad"/>
 </dependsOn>
 <dependsOn>
  <name>Messenger_Stub</name>
  <referencedArtifact href="Messenger_Stub.iad"/>
 </dependsOn>
 <execParameter>
  <name>entryPoint</name>
  <value>
   <type>
    <kind>tk_string</kind>
   </type>
   <value>
    <string>createMessengerHome_Servant</string>
   </value>
  </value>
 </execParameter>
</Deployment:ImplementationArtifactDescription>
```

The `<label>`, `<location>`, and first `<dependsOn>` elements contain 
similar information to that in the Messenger_stub.iad file. However, the 
Messenger_svnt library also depends on the Messenger_stub library, as
reflected in the additional <dependsOn> element. In addition, the Messenger_svnt library has an entry point function, configured through the <execParameter> element. An entry point function always has the <name> of “entryPoint”. The execution parameter’s <value> element is actually an XML representation of a CORBA Any. The <value> element’s string value matches the name of the Messenger’s library entry point function as generated by the CIDL compiler. Additional information on the CIDL compiler is available in 32.5.

Note CIAO looks for an “entryPoint” execution parameter for any implementation artifact that ends in _svnt or _exec. Thus, your servant and executor implementation artifacts should end in _svnt and _exec, respectively.

The Messenger_exec Implementation Artifact Descriptor is nearly identical to the Messenger_svnt descriptor, as is shown below:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Executor Artifact</label>
  <location>Messenger_exec</location>
  <dependsOn>
    <name>ACE/TAO/CIAO</name>
    <referencedArtifact href="Libraries.iad"/>
  </dependsOn>
  <dependsOn>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </dependsOn>
  <execParameter>
    <name>entryPoint</name>
    <value>
      <type>
        <kind>tk_string</kind>
      </type>
      <value>
        <string>createMessengerHome_Impl</string>
      </value>
    </value>
  </execParameter>
</Deployment:ImplementationArtifactDescription>
```
The Messenger_exec implementation artifact also has a library entry point function, matching the name of the entry point function generated by the CIDL compiler.

In summary, each of the component’s Implementation Artifact Descriptor files contains information about one of the component’s libraries. There is an Implementation Artifact Descriptor for each of the Messenger_stub, Messenger_svnt, and Messenger_exec libraries.

**Messenger Component - CORBA Component Descriptor (.ccd)**

The *CORBA Component Descriptor* describes the component’s IDL3 interface in an XML format. Primarily, it describes the component’s exposed ports and attributes. The mapping from a component’s IDL file to a CORBA Component Descriptor is purely mechanical.

There are six kinds of component ports: Facet, SimplexReceptacle, MultiplexReceptacle, EventPublisher, EventEmitter, and EventConsumer, as shown in Table 32-4:

**Table 32-4 Component Port Types**

<table>
<thead>
<tr>
<th>Port Kind</th>
<th>Sample IDL3 Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facet</td>
<td>provides Runnable control</td>
</tr>
<tr>
<td>SimplexReceptacle</td>
<td>uses Runnable control</td>
</tr>
<tr>
<td>MultiplexReceptacle</td>
<td>uses multiple Runnable controls</td>
</tr>
<tr>
<td>EventPublisher</td>
<td>publishes Message message_publisher</td>
</tr>
<tr>
<td>EventEmitter</td>
<td>emits Message message_emitter</td>
</tr>
<tr>
<td>EventConsumer</td>
<td>consumes Message message_consumer</td>
</tr>
</tbody>
</table>

Recall that the Messenger component’s IDL3 is as follows:

```idl3
component Messenger {
    attribute string subject;

    provides Runnable control;
    provides Publication content;

    publishes Message message_publisher;
    provides History message_history;
};
```

We create theMessenger.ccd file describing the Messenger’s IDL3 interface as follows. Comments are interspersed.
The optional <label> element contains a description that may be used by a tool for display purposes. The <specificType> element contains the Interface Repository Id of the component’s IDL interface.

The descriptor has a <supportedType> element for the component’s Interface Repository Id. It also has a <supportedType> element for each IDL2 interface supported by the component either directly or through inheritance. A component may indicate that it supports an IDL2 interface through the supports keyword. For example.

```
component Messenger supports MyInterface {

would map to an additional <supportedType> element such as this:

<supportedType>IDL:MyInterface:1.0</supportedType>

A component supporting an interface inherits the operations, attributes, etc. from that interface. However, we don’t use the supports keyword in our examples.

The optional <idlFile> element points to the IDL file that is the source of this information. It is for documentation purposes.

```

A <property> element describes each of the component’s IDL attributes. This <property> element describes the component’s subject attribute. The
<name> element’s value matches the attribute name in the IDL file. The <type> element’s <kind> is a type code. In our example, the attribute is a string. The “Deployment and Configuration” specification (OMG Document ptc/03-07-08) and the Deployment.xsd schema file contain more information on representing data types in XML descriptors.

```
<port>
  <name>control</name>
  <exclusiveProvider>false</exclusiveProvider>
  <exclusiveUser>false</exclusiveUser>
  <optional>true</optional>
  <provider>true</provider>
  <specificType>IDL:Runnable:1.0</specificType>
  <supportedType>IDL:Runnable:1.0</supportedType>
  <kind>Facet</kind>
</port>
```

Each provides, uses, uses multiple, publishes, emits, and consumes declaration has a matching <port> element. This port corresponds to the provides Runnable control;

The <name> element’s value matches the facet name in the IDL file. The <specificType> element contains the Interface Repository Id of the facet’s IDL interface. There may be several <supportedType> elements; there is one for the facet’s most specific IDL interface and one for each inherited interface regardless of whether the inheritance is direct or indirect.

The <kind> element’s value is Facet. Valid <kind> values are Facet, SimplexReceptacle, MultiplexReceptacle, EventPublisher, EventEmitter, and EventConsumer. The <optional> element indicates if connecting to the port is optional or mandatory. The <provider> element’s value is true for provides and consumes, false for uses and publishes.

```
<port>
  <name>content</name>
  <exclusiveProvider>false</exclusiveProvider>
  <exclusiveUser>false</exclusiveUser>
  <optional>true</optional>
  <provider>true</provider>
  <supportedType>IDL:Publication:1.0</supportedType>
  <specificType>IDL:Publication:1.0</specificType>
  <kind>Facet</kind>
</port>
```
This is the Publication facet called content. Its declaration is nearly identical to that of the Runnable facet.

```
<port>
    <name>message_publisher</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <provider>false</provider>
    <supportedType>IDL:Message:1.0</supportedType>
    <specificType>IDL:Message:1.0</specificType>
    <kind>EventPublisher</kind>
</port>
```

This is the Message publishing port called message_publisher. The <supportedType> and <specificType> are the Interface Repository Id of the event type being published. The port’s <kind> is EventPublisher.

```
<port>
    <name>message_history</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>false</exclusiveUser>
    <optional>true</optional>
    <provider>true</provider>
    <supportedType>IDL:History:1.0</supportedType>
    <specificType>IDL:History:1.0</specificType>
    <kind>Facet</kind>
</port>
```

This is the History facet called message_history. Its declaration is nearly identical to that of the to the Runnable and Publication facets.

```
<configProperty>
    <name>subject</name>
    <value>
        <type>
            <kind>tk_string</kind>
        </type>
        <string>Default Subject</string>
    </value>
</configProperty>
```
This `<configProperty>` element sets a default value for the Messenger’s subject attribute. Both the “Deployment and Configuration” specification and the `Deployment.xsd` schema file contain more information on representing data types and values in XML descriptors.

**Note**  
*CIAO currently ignores `<configProperty>` elements that set values for IDL attributes.*

The descriptor also recognizes one or more optional `<infoProperty>` elements that provide non-functional information that might be displayed by a tool as explained above.

In summary, the Messenger’s CORBA Component Descriptor file, `Messenger.ccd`, is an XML rendition of the component’s IDL3 interface. It contains a `<property>` element for each component attribute and a `<port>` element for each port.

**Messenger Component - Component Implementation Descriptor (.cid)**

The Messenger’s Component Implementation Descriptor describes the monolithic implementation of the Messenger component. A monolithic implementation consists of a set of implementation artifacts, or libraries. (By contrast, an assembly implementation is a component implementation that consists of subcomponents). Our monolithic Messenger implementation pulls together the Messenger’s three dynamic libraries—`Messenger_stub`, `Messenger_svnt`, and `Messenger_exec`.

The Messenger component’s Component Implementation Descriptor follows. Comments are interspersed.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentImplementationDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
 <label>Messenger Implementation</label>
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
The optional `<label>` element describes the implementation. A tool may use it for display purposes.

```xml
<implements href="Messenger.ccd"/>
```

The `<implements>` element describes the interface that the component implements by referencing the component’s CORBA Component Descriptor file.

```xml
<monolithicImpl>
  <primaryArtifact>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </primaryArtifact>
  <primaryArtifact>
    <name>Messenger_Svnt</name>
    <referencedArtifact href="Messenger_Svnt.iad"/>
  </primaryArtifact>
  <primaryArtifact>
    <name>Messenger_Exec</name>
    <referencedArtifact href="Messenger_Exec.iad"/>
  </primaryArtifact>
</monolithicImpl>
```

The `<monolithicImpl>` element pulls together the Messenger’s three libraries. Each library is a `<primaryArtifact>` represented by a reference to an Implementation Artifact Descriptor file.

```xml
<configProperty>
  <name>ComponentIOR</name>
  <value>
    <type>
      <kind>tk_string</kind>
    </type>
    <string>Messenger.ior</string>
  </value>
</configProperty>
```

CIAO supports one optional `<configProperty>`, the `ComponentIOR` property, for a component implementation. At run time, the component server writes the component’s object reference to the file indicated by the `ComponentIOR` property value. By default, the component server writes the
file to the directory from which it was launched. A non-CCM CORBA client may use that IOR file to discover the component.

</Deployment:ComponentImplementationDescription>

The Component Implementation Descriptor also accepts <capability> elements which can be used by the component server to choose between component implementations. It also accepts non-functional <infoProperty> elements as explained above.

In summary, the Messenger’s Component Implementation Descriptor constructs a monolithic Messenger implementation from the Messenger’s libraries.

**Messenger Component - Component Package Descriptor (.cpd)**

The Component Package Descriptor is the component’s top-level packaging file. It can describe multiple alternative implementations of the same component interface and can contain configuration properties for the component.

In our example, we merely reference the component implementation defined in the previous subsection.

The Component Package Descriptor for the Messenger component is as follows, with comments interspersed:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentPackageDescription
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">

<label>Messenger Component</label>

The optional <label> element is a human readable package label that may be used by a tool for display purposes.

<realizes href="Messenger.ccd"/>

The <realizes> element indicates the component’s IDL3 interface by referencing the component’s CORBA Component Descriptor file.
The `<implementation>` element references one or more Component Implementations Descriptors. Our example has just one Messenger implementation, so we refer to that implementation here. A more complex example may have multiple implementations and may use `<deployRequirement>` elements to enable a component server to choose between them.

```xml
</Deployment:ComponentPackageDescription>
```

The Component Package Descriptor also recognizes `<infoProperty>` documentation elements and `<configProperty>` default attribute value configuration elements.

---

**Note**  
*CIAO does not yet support the setting of a default attribute value through a `<configProperty>` element, so this value is currently ignored.*

---

To summarize, the Component Package Descriptor is the component’s top-level descriptor, representing the component to the rest of the application.

**Messenger Component - Summary**  
The table summarizes the six Messenger component descriptor files

**Table 32-5 : Messenger Descriptor Files**

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messenger_stub.iad</td>
<td>Implementation Artifact Descriptor for the Messenger’s stub library</td>
</tr>
<tr>
<td>Messenger_svnt.iad</td>
<td>Implementation Artifact Descriptor for the Messenger’s servant library</td>
</tr>
<tr>
<td>Messenger_exec.iad</td>
<td>Implementation Artifact Descriptor for the Messenger’s executor library</td>
</tr>
<tr>
<td>Messenger.ccd</td>
<td>CORBA Component Descriptor for the Messenger’s IDL3 interface</td>
</tr>
<tr>
<td>Messenger.cid</td>
<td>Component Implementation Descriptor describing the Messenger’s implementation in terms of its libraries</td>
</tr>
</tbody>
</table>
32.2.5.5 Receiver Component Descriptors

The Receiver component type has a similar set of descriptor files, as illustrated by the diagram.
The primary differences between the Receiver’s and the Messenger’s descriptor files are in the Implementation Artifact Descriptor and the CORBA Component Descriptor.

The Receiver’s three Implementation Artifact Descriptors reflect the fact that each of the Receiver’s three libraries has a dependency on the Messenger’s stub library. The dependency is illustrated by the Receiver_stub.iad file:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ImplementationArtifactDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Receiver Stub Artifact</label>
  <location>Receiver_stub</location>
  <dependsOn>
    <name>ACE/TAO/CIAO</name>
    <referencedArtifact href="Libraries.iad"/>
  </dependsOn>
  <dependsOn>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </dependsOn>
</Deployment:ImplementationArtifactDescription>
```

The Receiver’s servant and executor Implementation Artifact Descriptor files are analogous to the Messenger’s with the additional dependency on the Messenger_stub library. We do not show them here.

The Receiver’s CORBA Component Descriptor file describes the Receiver’s IDL3 interface, which is as follows:

```java
component Receiver {
    consumes Message message_consumer;
    uses History message_history;
}
```

The CORBA Component Descriptor describes the Receiver’s two ports, an EventConsumer and a SimplexReceptacle. It is as follows, with comments interspersed:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentInterfaceDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Receiver Stub Artifact</label>
  <location>Receiver_stub</location>
  <dependsOn>
    <name>ACE/TAO/CIAO</name>
    <referencedArtifact href="Libraries.iad"/>
  </dependsOn>
  <dependsOn>
    <name>Messenger_Stub</name>
    <referencedArtifact href="Messenger_Stub.iad"/>
  </dependsOn>
</Deployment:ComponentInterfaceDescription>
```
32.2 Example - The Messenger Application

The `message_consumer` port is an `EventConsumer` that consumes `Message` events. The `<supportedType>` and `<specificType>` elements contain the Interface Repository Id of the published event type.

The `message_history` port uses the `Messenger`'s `History` interface. The `uses` keyword in the component’s interface indicates that it is a `SimplexReceptacle`, meaning that the receptacle connects to exactly one `History` facet. The `message_consumer` port may receive `Message` events from multiple publishers, but the `message_history` port may only retrieve the `History` from one provider.
The table summarizes the six Receiver descriptor files.

**Table 32-6 Receiver Descriptor Files**

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver_stub.iad</td>
<td>Implementation Artifact Descriptor for the Receiver’s stub library</td>
</tr>
<tr>
<td>Receiver_svnt.iad</td>
<td>Implementation Artifact Descriptor for the Receiver’s servant library</td>
</tr>
<tr>
<td>Receiver_exec.iad</td>
<td>Implementation Artifact Descriptor for the Receiver’s executor library</td>
</tr>
<tr>
<td>Receiver.ccd</td>
<td>CORBA Component Descriptor for the Receiver’s IDL3 interface</td>
</tr>
<tr>
<td>Receiver.cid</td>
<td>Component Implementation Descriptor describing the Receiver’s implementation in terms of its libraries</td>
</tr>
<tr>
<td>Receiver.cpd</td>
<td>Component Package Descriptor packaging the Receiver component into one deployable package.</td>
</tr>
</tbody>
</table>
32.2.5.6 Administrator Component Descriptors

The Administrator component type also has a similar set of descriptor files, as illustrated in the diagram.

![Diagram of Administrator Component Descriptors](image)

Figure 32-23 Administrator Descriptor Files

Like the Receiver, each of the Administrator’s libraries depends on the Messenger’s Messenger_stub library. We won’t replicate the Implementation Artifact Descriptor files here.

The Administrator’s CORBA Component Descriptor file describes the Administrator’s IDL3 interface, which is as follows:

```idl3
component Administrator {
```
CIAO and CCM

uses multiple Runnable runnables;
uses multiple Publication content;

The Administrator’s CORBA Component Descriptor describes the two Administrator MultiplexReceptacle ports. It is as follows, with comments interspersed:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentInterfaceDescription
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Administrator Component</label>
  <specificType>IDL:Administrator:1.0</specificType>
  <supportedType>IDL:Administrator:1.0</supportedType>
  <idlFile>Administrator.idl</idlFile>
  <port>
    <name>runnables</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true</exclusiveUser>
    <optional>true</optional>
    <provider>false</provider>
    <supportedType>IDL:Runnable:1.0</supportedType>
    <specificType>IDL:Runnable:1.0</specificType>
    <kind>MultiplexReceptacle</kind>
  </port>

  <port>
    <name>content</name>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true</exclusiveUser>
    <optional>true</optional>
    <provider>false</provider>
    <supportedType>IDL:Publication:1.0</supportedType>
    <specificType>IDL:Publication:1.0</specificType>
    <kind>MultiplexReceptacle</kind>
  </port>
</Deployment:ComponentInterfaceDescription>
```

The runnables port uses the Messenger’s Runnable interface. The uses multiple keyword in the IDL3 interface indicates that it is a MultiplexReceptacle, meaning that it may connect to many Runnable facets.
The content port uses the Messenger’s Publication interface. It is also a MultiplexReceptacle, meaning that it may connect to many Publication facets.

The table summarizes the six Administrator descriptor files.

**Table 32-7 Administrator Descriptor Files**

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administrator_stub.iad</td>
<td>Implementation Artifact Descriptor for the Administrator’s stub library</td>
</tr>
<tr>
<td>Administrator_svnt.iad</td>
<td>Implementation Artifact Descriptor for the Administrator’s servant library</td>
</tr>
<tr>
<td>Administrator_exec.iad</td>
<td>Implementation Artifact Descriptor for the Administrator’s executor library</td>
</tr>
<tr>
<td>Administrator.ccd</td>
<td>CORBA Component Descriptor for the Administrator’s IDL3 interface</td>
</tr>
<tr>
<td>Administrator.cid</td>
<td>Component Implementation Descriptor describing the Administrator’s implementation in terms of its libraries</td>
</tr>
<tr>
<td>Administrator.cpd</td>
<td>Component Package Descriptor packaging the Administrator component into one deployable package.</td>
</tr>
</tbody>
</table>
32.2.5.7 Messenger Assembly Descriptors

An assembly is a component implementation that consists of a set of subcomponent instances connected through their ports.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
- Deploy each component into a component container
- Build the application
- Run the application

Figure 32-24 Road Map

We package the Messenger, Receiver, and Application components into one top-level component we refer to as the Messenger Assembly. Our Messenger Assembly consists of one Messenger component instance, two Receiver component instances, and one Administrator component instance.

The Messenger Assembly’s deployment is described by three descriptor files: a CORBA Component Descriptor describing the assembly’s exposed properties and ports; a Component Implementation Descriptor describing the assembly’s implementation in terms of its subcomponent instances and the connections between them; and a Component Package Descriptor that
packages the assembly into a deployable component. The relationships between the descriptor files are illustrated in the diagram.

A **CORBA Component Descriptor** describes the Messenger Assembly’s public interface. An assembly may expose ports and attributes of its subcomponents to the outside world. The Messenger Assembly exposes the Messenger component’s subject attribute, but does not expose any Messenger, Receiver, or Administrator ports.

A **Component Implementation Descriptor** describes the Messenger Assembly’s implementation. Its implementation is composed of one instance of the Messenger component, two instances of the Receiver component, and one instance of the Administrator component. The Component Implementation Descriptor describes how the component instances’ facets and event publishers connect to receptacles and event consumers to comprise the assembly.
A Component Package Descriptor can describe many alternate implementations of the assembly. Our example provides one implementation of the Messenger Assembly.

**Messenger Assembly - CORBA Component Descriptor**

A CORBA Component Descriptor describes a component’s ports and attributes. An assembled component may expose ports or attributes of its subcomponents. Our Messenger Assembly merely exposes the Messenger component’s subject attribute.

The Messenger Assembly’s CORBA Component Descriptor follows.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentInterfaceDescription
  xmlns:Deployment="http://www.omg.org/Deployment"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Assembly</label>
  <property>
    <name>subject</name>
    <type>
      <kind>tk_string</kind>
    </type>
  </property>
</Deployment:ComponentInterfaceDescription>
```

The `<property>` element indicates that the assembly exposes an attribute called “subject”, whose type is a string. The assembly’s Component Implementation Descriptor file, described in the next section, defines how the assembly’s subject attribute is mapped to the subject attribute of its Messenger subcomponent.

We can think of the Messenger Assembly as a component whose implied IDL3 interface is the following:

```idl3
component MessengerAssembly {
  attribute string subject;
};
```
**Messenger Assembly - Component Implementation Descriptor**

The Messenger Assembly’s Component Implementation Descriptor describes the subcomponent instances that comprise the assembly and the connections between their ports. The Component Implementation Descriptor describes the subcomponent instances and connections as shown in the deployment diagram.

![Messenger Assembly Diagram](image)

**Figure 32-26 Messenger Application Deployment Diagram**

The Messenger Assembly’s Component Implementation Descriptor is as follows, with comments interspersed.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentImplementationDescription
 xmlns:Deployment="http://www.omg.org/Deployment">
```

---

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University. Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
The `<implements>` element references the Messenger Assembly’s CORBA Component Descriptor documented in the previous section.

An `<assemblyImpl>` element indicates that this is an assembly-based component, meaning that is composed of subcomponent instances.

The `<instance>` elements create the Messenger instance, the two Receiver instances, and the Administrator instance. Each `<instance>` refers to the Component Package Descriptor of its component type. The `xml:id` attributes of the instances are used by `<connection>` elements to connect the instances’ ports.
This connection connects the Messenger instance’s `message_publisher` port to one Receiver instance’s `message_consumer` port. The connection’s `<name>` is a unique identifier for the connection within the assembly. The value in each `<portName>` element must match the port name in the Messenger’s and Receiver’s CORBA Component Descriptor files. The `<instance>` element’s `xml:idref` attribute matches the `<instance>` element’s `xml:id` attribute above.

This connection connects the Messenger’s `message_history` facet to a Receiver instance’s `message_history` receptacle.
The `message_publisher-to-message_consumer` connection and the `message_history-to-message_history` connection are illustrated by the highlights in the deployment diagram.

![Deployment Diagram](image)

**Figure 32-27 One Messenger and Receiver Connection**

```xml
<connection>
  <name>Messenger_to_Second_Receiver_Publisher</name>
  <internalEndpoint>
    <portName>message_publisher</portName>
    <instance xmlns:idref="a_Messenger"/>
  </internalEndpoint>
  <internalEndpoint>
    <portName>message_consumer</portName>
    <instance xmlns:idref="second_Receiver"/>
  </internalEndpoint>
</connection>
```
These two connections connect the second Receiver instance to the Messenger instance. The Messenger instance’s `message_publisher` port connects to the Receiver instance’s `message_consumer` port and the Messenger instance’s `message_history` facet connects to the Receiver instance’s `message_history` receptacle.

These two connections connect the Administrator instance to the Messenger instance. The Messenger instance’s `control` facet connects to the Administrator instance’s `runnables` receptacle and the Messenger instance’s `content` facet connects to the Administrator instance’s `content` receptacle.
The <externalProperty> element maps the Messenger Assembly’s exposed subject attribute to the Messenger component instance’s subject attribute. The Messenger Assembly doesn’t implement its own subject attribute; it must map to an attribute of one of its subcomponents.

In summary, the Messenger Assembly’s Component Implementation Descriptor describes the four subcomponent instances that comprise the assembly and describes the connections that connect their ports together.

**Messenger Assembly - Component Package Descriptor**

The Messenger Assembly’s Component Package Descriptor is the top-level descriptor that represents the assembly as a deployable component. That should sound familiar; the Messenger, Receiver, and Administrator Component Package Descriptors serve exactly the same purpose.

The Messenger Assembly’s Component Package Descriptor is nearly identical to the Messenger component’s.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentPackageDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
 <label>Messenger Assembly Package</label>
 <realizes href="MessengerAssembly.ccd"/>
 <implementation>
  <name>Messenger Application</name>
  <referencedImplementation href="MessengerAssembly.cid"/>
 </implementation>
</Deployment:ComponentPackageDescription>
```
The `<realizes>` element references the assembly’s CORBA Component Descriptor, which describes the assembly’s implied IDL3 interface. The `<referencedImplementation>` element references the assembly’s Component Implementation Descriptor, which describes the assembly’s implementation in terms of its subcomponents.

**Messenger Assembly - Summary**

The Messenger Assembly’s descriptors files describe the Messenger Assembly’s composition in terms of its subcomponent instances the connections between them. The table summarizes the Messenger Assembly descriptor files.

**Table 32-8 Messenger Assembly Descriptor Files**

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessengerAssembly.ccd</td>
<td>CORBA Component Descriptor for the Messenger Assembly’s implied IDL3 interface</td>
</tr>
<tr>
<td>MessengerAssembly.cid</td>
<td>Component Implementation Descriptor describing the Messenger Assembly’s implementation in terms of its subcomponent instances and connections</td>
</tr>
<tr>
<td>MessengerAssembly.cpd</td>
<td>Component Package Descriptor packaging the Messenger Assembly component into one deployable package.</td>
</tr>
</tbody>
</table>
32.2.5.8 Application Descriptors

The application’s deployment descriptors describe how the assembly’s component instances are deployed onto logical nodes and how each logical node is mapped to a physical component container.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
  ✪ **Deploy each component into a component container**
- Build the application
- Run the application

**Figure 32-28 Road Map**
The application’s UML Deployment Diagram illustrates the deployment.

Figure 32-29 Deployment of the Messenger Application

The Application’s deployment is described by five descriptor files: a Package Configuration Descriptor that wraps the Messenger Assembly’s package descriptor; a Top-level Package Descriptor that represents the entire application; a Component Deployment Plan descriptor that maps the Message Assembly’s subcomponent instances to logical deployment nodes; a Component Domain Descriptor that describes each of the logical nodes; and a node map that maps logical deployment nodes to physical component server processes.
The Package Configuration Descriptor describes one possible configuration of a component package by indicating deployment requirements and/or configuration properties of the component package.

The Top-level Package Descriptor references the Package Configuration Descriptor that is the root of the deployed application. It always has the name package.tpd.

The Component Deployment Plan contains the bulk of the application’s deployment information. It maps each component instance onto a logical deployment node, achieving a separation of concerns between the Component Implementation Descriptor’s instance connections and the Component Deployment Plan’s node mappings.

The Component Domain Descriptor describes the target deployment environment in terms of its nodes, interconnects, and bridges.

The Node Map maps each logical node onto a physical component server process.
The relationships between the application descriptor files are illustrated in the diagram.

![Diagram illustrating relationships between application descriptor files](image)

**Figure 32-30 The Application’s Deployment Descriptors**

**Application - Package Configuration Descriptor**

The Package Configuration Descriptor describes one configuration of a deployable component package. It may define deployment requirements or configure attribute values. Our Package Configuration Descriptor references the Messenger Assembly’s Component Package Descriptor and configures a default value for the Messenger Assembly’s exposed `subject` attribute. The descriptor is as follows, with comments interspersed:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:PackageConfiguration
xmlns:Deployment="http://www.omg.org/Deployment"
>

```
The `<basePackage>` element references the Messenger Assembly’s Component Package Descriptor. Either a `<basePackage>` element or a `<specializedConfig>` element is mandatory. A `<specializedConfig>` element can reference another Package Configuration Descriptor and override its requirements and/or configuration values.

The `<configProperty>` element defines a default value for the Messenger Assembly’s `subject` attribute.

**Note** CIAO does not yet support the setting of a default attribute value through a `<configProperty>` element, so this value is currently ignored.

A Package Configuration Descriptor may also contain `<selectRequirement>` requirement elements. In future implementations of CIAO, these elements would be matched against `<capability>` elements in the Component Implementation Description.

**Application - Top-level Package Descriptor**
Each application has exactly one Top-level Package Descriptor. It is always named `package.tpd`, and it points to the application’s Package...
Configuration Descriptor file. The Top-level Package Descriptor for our application is as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:TopLevelPackageDescription
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <package href="Application.pcd"/>
</Deployment:TopLevelPackageDescription>
```

A Top-level Package Descriptor has exactly one `<package>` element that points to the application’s Package Configuration Descriptor.

**Application - Component Deployment Plan**

The application’s Component Deployment Plan describes how the Messenger Assembly’s component instances are deployed onto logical processing nodes. Through this descriptor, the application deployer can vary the component instance-to-node mapping independently from the connections between component instances.

The Component Deployment Plan for our application is as follows, with comments interspersed:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:DeploymentPlan
 xmlns:Deployment="http://www.omg.org/Deployment"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Deployment Plan</label>
  <instance xmi:id="Messenger_Instance_ID">
    <name>Messenger_Instance</name>
    <node>Messenger_Node</node>
  </instance>
  <instance xmi:id="First_Receiver_Instance_ID">
    <name>First_Receiver_Instance</name>
  </instance>
</Deployment:DeploymentPlan>
```

This `<instance>` element indicates that the Messenger_Instance is deployed onto the Messenger_Node. The value of the `<name>` element must match the Messenger instance’s `<name>` element in the Messenger Assembly’s Component Implementation Descriptor file.
These two `<instance>` elements indicate that each of the two Receiver instances is deployed on its own logical node. Again, the value of each `<name>` element must match the Receiver instances’ `<name>` elements in the Messenger Assembly’s Component Implementation Descriptor file.

This `<instance>` element indicates that the Administrator instance is deployed onto the Administrator_Node.

In summary, the Component Deployment Plan maps each component instance onto a logical processing node. Each instance name must match an instance name in the assembly’s Component Implementation Descriptor.

**Application - Component Domain Descriptor**

The Component Domain Descriptor describes the target environment in terms of its nodes, interconnects, and bridges. The Messenger application’s Component Domain Descriptor is as follows:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Deployment:Domain xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
  <label>Messenger Application Domain</label>
  <node>
    <name>Messenger_Node</name>
    <label>Messenger's Node</label>
  </node>
  <node>
    <name>First_Receiver_Node</name>
  </node>
</Deployment:Domain>
```
This Component Domain Descriptor describes the four nodes of the Messenger application. The `<name>` of each `<node>` matches the node name in the Component Deployment Plan. The `<label>`, as always, is optional and may be used for display purposes by a tool. Our sample application doesn’t use this descriptor, but we define it for completeness.

**Node Map**

The node map is a text file that maps each of the Component Deployment Plan’s logical nodes onto a physical component server process by mapping each logical node to a `NodeManager` object reference. The node map for the Messenger application is as follows:

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administrator_Node</td>
<td>corbaloc:iiop:localhost:10000/NodeManager</td>
</tr>
<tr>
<td>First_Receiver_Node</td>
<td>corbaloc:iiop:localhost:20000/NodeManager</td>
</tr>
<tr>
<td>Second_Receiver_Node</td>
<td>corbaloc:iiop:localhost:30000/NodeManager</td>
</tr>
<tr>
<td>Messenger_Node</td>
<td>corbaloc:iiop:localhost:40000/NodeManager</td>
</tr>
</tbody>
</table>

The contents of this file determine where each component executes. Our deployment environment consists of four `NodeManager` processes running on the `localhost`, each listening on a different port. Each `NodeManager` is a component server, capable of dynamically loading a component’s libraries and making connections between components. The `NodeManager` is documented in 32.2.7 and 32.4.

The node map enables a great deal of deployment flexibility. We could deploy the Messenger application across a network simply by running our `NodeManager` processes across the network and changing our node map’s `NodeManager` object references to reflect that.

We could also deploy several component instances on one component server simply by mapping several logical nodes to the same `NodeManager` object reference.
Messenger Application - Summary
The Messenger Application’s descriptors describe how each subcomponent instance is deployed onto a physical component server process. The table summarizes the Messenger application’s descriptor files.

Table 32-9 Messenger Application Descriptor Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application.pcd</td>
<td>Package Configuration Descriptor that configures the top-level component’s deployment attributes.</td>
</tr>
<tr>
<td>package.tpd</td>
<td>Top-level Package Descriptor that represents the application.</td>
</tr>
<tr>
<td>Application.cdp</td>
<td>Component Deployment Plan that maps component instances to logical nodes.</td>
</tr>
<tr>
<td>Domain.cdd</td>
<td>Component Domain Descriptor that describes the target deployment environment.</td>
</tr>
<tr>
<td>ApplicationNodeMap.dat</td>
<td>Text file that maps each logical node to a physical component server object.</td>
</tr>
</tbody>
</table>

The following sections discusses the execution of the Messenger application.

32.2.6 Building the Messenger Application
The Messenger application consists of three component types: Messenger, Receiver, and Administrator. Each component type is composed of three libraries: a stub library, a servant library, and an executor library. We manage
these libraries with three Make Project Creator (MPC) files, one file for each component type.

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- **Build the application**
- Run the application

Figure 32-31 Road Map

The example’s source code, build files, and XML descriptor files are in the 
$TAO_ROOT/DevGuideExamples/CIAO/Messenger directory.

32.2.6.1 Setting Up Your Environment

There are several environment variables used by ACE, TAO, and CIAO during both the compilation and execution of applications. Information about ACE and TAO environment variables is available at 3.2. CIAO’s environment variables are described below. Syntax for Windows is shown in parentheses.

- CIAO_ROOT
  The base path for all CIAO-related code, normally $TAO_ROOT/CIAO (%TAO_ROOT%CIAO).
- XERCESCROOT
  The base directory of the Xerces C++ installation. See 32.3.1 and 32.3.2 for more information on Xerces C++.
- Library Path
  The library path must include the directory containing the Xerces C++ dynamic libraries, $XERCESCROOT/bin (%XERCESCROOT%bin). You should add this location to your LD_LIBRARY_PATH environment variable.
or its equivalent. (On Windows, add this directory to your PATH so DLLs can be located at run time.)

### 32.2.6.2 Creating the Messenger’s MPC File

The CIAO source tree contains a `generate_component_mpc.pl` script to generate the beginning of a component’s MPC file.

**UNIX and UNIX-like Systems**
The script is `$CIAO_ROOT/bin/generate_component_mpc.pl`

**Windows Systems**
The script is `%CIAO_ROOT%\bin\generate_component_mpc.pl`.

**General Usage**
The general usage of the `generate_component_mpc.pl` script is as follows:

```
$CIAO_ROOT/bin/generate_component_mpc.pl <component name>
```

For example:

```
$CIAO_ROOT/bin/generate_component_mpc.pl Messenger
```

creates an MPC file called `Messenger.mpc`.

The script also prints the following text to suggest that you generate an export header file for each Messenger library:

```
Run the following commands also:
  generate_export_file.pl MESSENGER_STUB > Messenger_stub_export.h
  generate_export_file.pl MESSENGER_SVNT > Messenger_svnt_export.h
  generate_export_file.pl MESSENGER_EXEC > Messenger_exec_export.h
```

We deploy our component executors in dynamic libraries. On Windows platforms, classes exported from a dynamic library must define an export macro. On UNIX-like platforms, the export macros define to nothing. However, future versions of the gcc compiler will support the C++ export keyword, which may reduce code size by reducing the number of exported symbols. In either case, the export macros enable cross-platform development.
The `generate_export_file.pl` script is in the $ACE_ROOT/bin directory on UNIX-like systems and in the %ACE_ROOT%\bin directory on Windows systems. Run the script as follows:

```
generate_export_file.pl MESSENGER_STUB > Messenger_stub_export.h
generate_export_file.pl MESSENGER_SVNT > Messenger_svnt_export.h
generate_export_file.pl MESSENGER_EXEC > Messenger_exec_export.h
```

See 5.12 for more information on the `generate_export_file.pl` script.

**The Messenger's MPC File**

The generated MPC file is as follows:

```
project(Messenger_stub): ciao_client {

    sharedname = Messenger_stub
    idlflags += -Wb,stub_export_macro=MESSENGER_STUB_Export
    idlflags += -Wb,stub_export_include=Messenger_stub_export.h
    idlflags += -Wb,skel_export_macro=MESSENGER_SVNT_Export
    idlflags += -Wb,skel_export_include=Messenger_svnt_export.h
    dynamicflags   = MESSENGER_STUB_BUILD_DLL

    IDL_Files {
        Messenger.idl
    }

    Source_Files {
        MessengerC.cpp
    }
}

project(Messenger_svnt) : ciao_servant {

    after += Messenger_stub
    sharedname = Messenger_svnt
    libs += Messenger_stub
    idlflags += -Wb,export_macro=MESSENGER_SVNT_Export
    idlflags += -Wb,export_include=Messenger_svnt_export.h
    dynamicflags = MESSENGER_SVNT_BUILD_DLL

    CIDL_Files {
        Messenger.cidl
    }

    IDL_Files {
        MessengerE.idl
    }
```
That’s a reasonable start towards our final Messenger.mpc file. We edit the file as follows, with comments interspersed. First, we discuss the project for the Messenger’s stub library, Messenger_stub.

```cpp
project(Messenger_stub): ciao_client_dnc {

Because we deploy the application using the DAnCE facility, we change the ciao_client base project dependency to ciao_client_dnc.

```cpp
requires += cidl
requires += exceptions
```

This project requires the CIDL compiler. It also requires native C++ exception support.

```cpp
sharedname = Messenger_stub
idlflags += -Wb,stub_export_macro=MESSENGER_STUB_Export
idlflags += -Wb,stub_export_include=Messenger_stub_export.h
idlflags += -Wb,skel_export_macro=MESSENGER_SVNT_Export
idlflags += -Wb,skel_export_include=Messenger_svnt_export.h
dynamicflags = MESSENGER_STUB_BUILD_DLL
```

We make no changes to the sharedname, idlflags, or dynamicflags.
IDL_Files {
    Runnable.idl
    Publication.idl
    Message.idl
    History.idl
    Messenger.idl
}

The Messenger component’s interfaces, event types, and component declaration are spread across five IDL files. The generate_component_mpc.pl script does not know this. Thus, we add four IDL files to the IDL_Files section.

Source_Files {
    RunnableC.cpp
    PublicationC.cpp
    MessageC.cpp
    HistoryC.cpp
    MessengerC.cpp
}

We add stub source code files for each of the Messenger component’s IDL files.

Next, we discuss the project for the Messenger’s servant library, Messenger_svnt.

project(Messenger_svnt): ciao_servant_dnc {

Again, because we deploy the application using the DAnCE facility, we change the ciao_servant base project dependency to ciao_servant_dnc.

    requires += cidl
    requires += exceptions

This project also requires the CIDL compiler and native C++ exception support.

    after += Messenger_stub
    sharedname = Messenger_svnt
    libs += Messenger_stub
idlflags += -Wb,export_macro=MESSENGER_SVNT_Export
idlflags += -Wb,export_include=Messenger_svnt_export.h
dynamicflags = MESSENGER_SVNT_BUILD_DLL

We make no changes to the after, sharedname, libs, idlflags, or
dynamicflags.

// cidlc does NOT automatically add the current directory to
// the include path. This is a workaround to add it. We have
// to insert it before the "--" that is at the end of the
// default cidlflags.
cidlflags -= --
cidlflags += -I. --

Our IDL and CIDL files require that the current directory is in the CIDL
compiler’s include path. These two lines add the current directory to the CIDL
compiler’s include path.

// project must be a ciao_servant or ciao_server to use CIDL files
CIDL_Files {
   Messenger.cidl
}

IDL_Files {
   MessengerE.idl
}

We make no changes to the CIDL_Files or the IDL_Files.

Source_Files {
    RunnableS.cpp
    PublicationS.cpp
    MessageS.cpp
    HistoryS.cpp
    MessengerS.cpp
    MessengerEC.cpp
    Messenger_svnt.cpp
}

We add skeleton source code files for the Messenger component’s IDL files.

}

Finally, we discuss the project for the Messenger’s executor library,
Messenger_exec.
32.2 Example - The Messenger Application

project(Messenger_exec): ciao_component_dnc {

Once more, because we deploy the application using the DAnCE facility, we change the ciao_component base project dependency to ciao_component_dnc.

requires += cidl
requires += exceptions

This project also requires the CIDL compiler and native C++ exception support.

after += Messenger_svnt
sharedname = Messenger_exec
libs += Messenger_stub Messenger_svnt
idlflags += -Wb,export_macro=MESSENGER_EXEC_Export
idlflags += -Wb,export_include=Messenger_exec_export.h
dynamicflags = MESSENGER_EXEC_BUILD_DLL

We make no changes to the after, sharedname, libs, idlflags, or dynamicflags.

IDL_Files {
}

Source_Files {
    MessengerES.cpp
    Messenger_exec_i.cpp
    Publication_exec_i.cpp
    History_exec_i.cpp
    Runnable_exec_i.cpp
}

We make quite a few changes to the executor library’s Source_Files section. First, we add MessengerES.cpp, the Messenger executor’s skeleton file. Then we change Messenger_exec.cpp to Messenger_exec_i.cpp to reflect the fact that we’ve renamed the CIDL-generated executor implementation file as described in 32.2.3. Finally, we add the facet executor implementation files.
### 32.2.6.3 Creating the Administrator’s and Receiver’s MPC Files

The Receiver’s and Administrator’s MPC files are similar to the Messenger’s. We generate each file using the `generate_component_mpc.pl` script.

```bash
generate_component_mpc.pl Receiver
generate_component_mpc.pl Administrator
```

We modify the generated MPC files by hand, just as we did for the Messenger. We’ll examine the modified `Receiver.mpc` file and highlight significant differences between `Receiver.mpc` and `Messenger.mpc`. Comments are interspersed. First, we discuss the project for the Receiver’s stub library, `Receiver_stub`.

```ciao
project(Receiver_stub): ciao_client_dnc {
  requires += cidl
  requires += exceptions

  after += Messenger_stub
  sharedname  = Receiver_stub
  libs += Messenger_stub
}
```

The Receiver’s stub library is dependent on the Messenger’s stub library and must be built after the Messenger’s stub library.

```ciao
idlflags += -Wb,stub_export_macro=RECEIVER_STUB_Export
idlflags += -Wb,stub_export_include=Receiver_stub_export.h
idlflags += -Wb,skel_export_macro=RECEIVER_SVNT_Export
idlflags += -Wb,skel_export_include=Receiver_svnt_export.h
dynamicflags = RECEIVER_STUB_BUILD_DLL

IDL_Files {
  Receiver.idl
}

Source_Files {
  ReceiverC.cpp
}
}
```

Next, we discuss the project for the Receiver’s servant library, `Receiver_svnt`.

```ciao
project(Receiver_svnt): ciao_servant_dnc {
  requires += cidl
  requires += exceptions
```
after += Receiver_stub \texttt{Messenger_svnt}
sharedname  = Receiver_svnt
libs += Receiver_stub \texttt{Messenger_stub} \texttt{Messenger_svnt}

The Receiver’s servant library is dependent on the Messenger’s stub and servant libraries and must be built after the Messenger’s servant library.

\begin{verbatim}
idlflags += -Wb,export_macro=RECEIVER_SVNT_Export
idlflags += -Wb,export_include=Receiver_svnt_export.h
dynamicflags = RECEIVER_SVNT_BUILD_DLL

// cidlc does NOT automatically add the current directory to
// the include path. This is a workaround to add it. We have
// to insert it before the "--" that is at the end of the
// default cidlflags.
cidlflags -= --
cidlflags += -I. --

CIDL_Files {
  Receiver.cidl
}

IDL_Files {
  ReceiverE.idl
}

Source_Files {
  ReceiverS.cpp
  ReceiverEC.cpp
  Receiver_svnt.cpp
}
\end{verbatim}

Finally, we discuss the project for the Receiver’s executor library, \texttt{Receiver_exec}.

\begin{verbatim}
project(Receiver_exec): ciao_component_dnc {
  requires += cidl
  requires += exceptions

  after += Receiver_svnt
  sharedname  = Receiver_exec
  libs += Receiver_stub Receiver_svnt \texttt{Messenger_stub}

The Receiver’s executor library is dependent on the Messenger’s stub library.
\end{verbatim}
IDLflags += -Wb,export_macro=RECEIVER_EXEC_Export
IDLflags += -Wb,export_include=Receiver_exec_export.h
dynamicflags = RECEIVER_EXEC_BUILD_DLL

IDL_Files {
}

Source_Files {
    ReceiverES.cpp
    Receiver_exec_i.cpp
}

The Receiver’s executor library has just one executor implementation file.}

The modified Administrator.mpc file is similar. The primary difference between the Administrator component and the Receiver component is that the Administrator is not an event consumer, and thus the Administrator’s servant library does not depend on the Messenger’s servant library.

32.2.6.4 Running MPC
Execute the Make Project Creator to generate the Messenger’s build files for your platform. All TAO Developer’s Guide examples require that the $TAO_ROOT/DevGuideExamples directory be in the MPC path. For example:

UNIX and UNIX-like Systems

cd $TAO_ROOT/DevGuideExamples/CIAO/Messenger
$ACE_ROOT/bin/mwc.pl -include $TAO_ROOT/DevGuideExamples -type gnuace

Windows Systems

cd %TAO_ROOT%\DevGuideExamples\CIAO\Messenger
perl %ACE_ROOT%\bin\mwc.pl -include %TAO_ROOT%\DevGuideExamples -type vc71

See Chapter 4 for more information on MPC.

32.2.6.5 Building
Build the Messenger application using the target build environment.
32.2.7 Running the Messenger Application

Finally, we can execute the application.

Figure 32-32 Road Map

- Define an IDL interface for each component and its facets
- Implement each component and its facets
  - Define each component’s composition
  - Implement a C++ executor for each component and facet
- Describe the application’s deployment
  - Describe each component’s libraries and ports
  - Connect component instances through their ports
  - Deploy each component into a component container
- Build the application

Run the application
Recall that we deploy the Messenger application on four nodes as illustrated in the diagram.

![Figure 32-33 Messenger Deployment](image)

Each component type -- the Messenger, Receiver, and Administrator -- consists of a set of dynamic libraries. We have not created any executables. Deployment descriptor files, as described in 32.2.5, define how the component instances are created and connected together and how those component libraries are deployed onto physical nodes.

### 32.2.7.1 Setting Up Your Environment

Please see 32.2.6.1 for information on setting up the environment variables required to execute a CIAO application.
32.2.7.2 DAnCE Executables

CIAO’s Deployment And Configuration Engine (DAnCE), which implements the OMG “Deployment and Configuration of Component-based Distributed Applications” specification (OMG Document ptc/03-07-08), contains a set of executables to dynamically load component libraries, create component instances, and make connections between them. The table summarizes the DAnCE executables that we’ll use to deploy the application. The executables are described in more detail in 32.4.

**Table 32-10 DAnCE Executables**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeManager</td>
<td>A daemon process that launches NodeApplication component servers</td>
</tr>
<tr>
<td>NodeApplication</td>
<td>The component server</td>
</tr>
<tr>
<td>ExecutionManager</td>
<td>A process that maps component instances to component servers</td>
</tr>
<tr>
<td>RepositoryManager</td>
<td>A process that parses a set of deployment descriptors and sends deployment information to the ExecutionManager</td>
</tr>
</tbody>
</table>

32.2.7.3 Deploying the Messenger with DAnCE

First, we run four Node Manager daemon processes. In our example, each Node Manager process executes on the `localhost`. The four processes listen on ports 10000, 20000, 30000, and 40000, respectively. Each Node Manager launches a Node Application process when an application is deployed upon it.

```bash
$CIAO_ROOT/DAnCE/NodeManager/Node_Daemon \  
-ORBListenEndpoints iiop://localhost:10000 \  
-s "$CIAO_ROOT/DAnCE/NodeApplication/NodeApplication"

$CIAO_ROOT/DAnCE/NodeManager/Node_Daemon \  
-ORBListenEndpoints iiop://localhost:20000 \  
-s "$CIAO_ROOT/DAnCE/NodeApplication/NodeApplication"

$CIAO_ROOT/DAnCE/NodeManager/Node_Daemon \  
-ORBListenEndpoints iiop://localhost:30000 \  
-s "$CIAO_ROOT/DAnCE/NodeApplication/NodeApplication"

$CIAO_ROOT/DAnCE/NodeManager/Node_Daemon \  
-ORBListenEndpoints iiop://localhost:40000 \  
-s "$CIAO_ROOT/DAnCE/NodeApplication/NodeApplication"
```
Then, we start an Execution Manager that reads the node map file ApplicationNodeMap.dat and writes its own object reference to a file called em.iol. The Execution Manager executable must be run in the directory containing the application’s deployment descriptors.

```
cd $TAO_ROOT/DevGuideExamples/CIAO/Messenger/descriptors
CIAO_ROOT/DAnCE/ExecutionManager/Execution_Manager \ 
-o em.iol -i ApplicationNodeMap.dat
```

Finally, we start a Repository Manager that connects to the Execution Manager and reads the application’s Top-level Package Descriptor and its Component Deployment Plan. The Repository Manager executable must also be run in the directory containing the application’s deployment descriptors.

```
cd $TAO_ROOT/DevGuideExamples/CIAO/Messenger/descriptors
CIAO_ROOT/DAnCE/RepositoryManager/executor \ 
-p package.tpd -d Application.cdp -k file://em.iol
```

The Execution Manager deploys a component instance in each of the Node Manager windows.

The Messenger does not begin publishing automatically. One of the Node Manager windows contains the Administrator component instance. That window displays the following menu:

```
What do you want to do to the Messenger(s)?
1. Start
2. Stop
3. Change Publication Period
4. Change Publication Text
Please enter a selection:
```

Use the menu to start and stop publishing and change attributes of the publication. A more industrial-strength application might launch a GUI of some kind. The Administrator’s menu illustrates that a component implementation can contain user interface elements.

### 32.2.7.4 Debugging

It can be a challenge to debug a component executor implementation. Thorough unit testing uncovers many problems before the component is tested in deployment. However, it may be necessary to debug a component in its component server process.
We launch a component executor in the debugger by changing the Node Manager’s launch command for its Node Application. It is easiest to illustrate with an example.

**UNIX and UNIX-like Systems**

```
$CIAO_ROOT/DAnCE/NodeManager/Node_Daemon
   -ORBListenEndpoints iiop://localhost:10000
   -d 180
   -s "gdb --args $CIAO_ROOT/DAnCE/NodeApplication/NodeApplication"
```

**Windows Systems**

```
%CIAO_ROOT%\DAnCE\NodeManager\Node_Daemon.exe
   -ORBListenEndpoints iiop://localhost:10000
   -d 180
   -s "devenv /debugexe %CIAO_ROOT%\DAnCE\NodeApplication\NodeApplication.exe"
```

The Node Manager’s launch command launches the Node Application process in the debugger. The `-d 180` command-line option delays the launching of the NodeApplication process by 180 seconds, or three minutes. In that three minutes you must set a breakpoint in your component executor to stop program execution. Once a breakpoint is reached you may debug as usual.

### 32.3 Building CIAO

Before building CIAO, build the ACE and TAO libraries as described in Chapter 2 and Appendix A.

CIAO requires three external libraries: Xerces C++; Boost; and Utility. The DAnCE deployment framework requires Xerces C++. The CIDL compiler requires Boost and Utility. These build instructions describe how to obtain and build those libraries. For more information, see `$CIAO_ROOT/CIAO-INSTALL.html`.

Building CIAO entails several steps:

1. Build ACE and TAO, including the following targets:

   ```
   ACE
   ACEXML_Parser
   TAO
   ```
2. Obtain and build the Xerces C++ library
3. Obtain and build the Boost library
4. Obtain the Utility library
5. Set up the build environment
6. Enable the CIDL compiler in MPC’s global features file.
7. Generate build files with MPC
8. Build CIAO’s CIDL compiler
9. Build CIAO’s libraries and DAnCE executables

**32.3.1 Building CIAO with Visual C++**

The CIAO libraries and DAnCE executables can be built with either Visual C++ 6 or Visual C++ 7.1. However, the CIDL compiler can only be built with Visual C++ 7.1. This section contain directions for building CIAO and the CIDL compiler with Visual C++ 7.1.

**Obtain and Build the Xerces C++ Library**

The source code for Xerces C++ can be obtained from [http://xml.apache.org/xerces-c/](http://xml.apache.org/xerces-c/). At publication time, the latest version of Xerces C++ was version 2.6. Download and unzip the Xerces C++ 2.6 source code. The remainder of this section assumes that Xerces C++ is installed in a directory called `C:\xerces-c-src-2_6_0`.

The Xerces C++ site also contains many prebuilt distributions of the Xerces C++ library. If you find a binary distribution that matches your platform and compiler then you can avoid building Xerces C++.

Set the `XERCESCROOT` environment variable to your root Xerces C++ directory, as follows:
Open the Visual Studio solution file called xerces-all.sln in the %XERCESCROOT%\Projects\Win32\VC7\xerces-all directory. If Visual Studio asks if you want to convert the projects to the current version of Visual Studio, answer “yes”. If there is a Visual Studio solution file for Visual Studio .NET (Visual C++ 7.1), use it instead. Build the XercesLib target.

The Xerces C++ project does not install its include files and libraries in the directories where CIAO is expecting them. We manually rectify this. First, we copy the Xerces C++ source files to an include directory. This copies both header and source files, but that is not a problem.

```
set XERCESCROOT=C:\xerces-c-src_2_6_0
```

Then, we create directories for the Xerces C++ libraries.

```
mkdir %XERCESCROOT%\lib
mkdir %XERCESCROOT%\bin
```

Finally, we copy the Xerces C++ libraries to the appropriate directories.

```
cd %XERCESCROOT%\lib
copy ..\Build\Win32\VC7\Debug\xerces-c_2D.lib
```

```
cd %XERCESCROOT%\bin
```

```
copy ..\Build\Win32\VC7\Debug\xerces-c_2_6D.dll
```

### Obtain and Build the Boost library

CIAO’s CIDL compiler uses the Boost regex and filesystem libraries and the spirit parser framework. The spirit parser framework consists only of header files.

CIAO’s Windows build requires Boost version 1.30.2. The Boost 1.30.2 source tree and the latest version of the Boost Jam build system can be downloaded from the Boost web site, <http://www.boost.org>.

Install Boost 1.30.2 and the latest Boost Jam in the directories of your choice. For this example, we assume that Boost 1.30.2 is installed in C:\Boost-1.30.2 and Boost Jam is installed in C:\Boost-Jam.
You can edit the Boost Jamfile in C:\Boost-1.30.2\Jamfile to limit the build to the filesystem and regex libraries. For example:

```bash
# Boost Jamfile
project-root ;
# please order by name to ease maintenance
subinclude libs/date_time/build ;
subinclude libs/filesystem/build ;
subinclude libs/python/build ;
subinclude libs/regex/build ;
subinclude libs/signals/build ;
subinclude libs/test/build ;
subinclude libs/thread/build ;
```

Build the Boost 1.30.2 regex and filesystem libraries with Boost Jam as follows:

```bash
cd C:\Boost-1.30.2
vsvars32.bat
C:\Boost-Jam\bjam.exe "-sTOOLS=vc7.1"
```

Create a directory called C:\Boost-1.30.2\lib. Copy the regex and filesystem library files to C:\Boost-1.30.2\lib and rename them so CIAO’s build can find them.

```bash
mkdir C:\Boost-1.30.2\lib
cd C:\Boost-1.30.2\lib

copy C:\Boost-1.30.2\libs\regex\build\bin\libboost_regex.lib\vc7.1\debug\runtime-link-dynamic\libboost_regex_debug.lib

copy C:\Boost-1.30.2\libs\filesystem\build\bin\libboost_filesystem.lib\vc7.1\debug\runtime-link-dynamic\libboost_filesystem.lib

rename libboost_regex_debug.lib boost_regex_debug.lib
rename libboost_filesystem.lib boost_filesystem_debug.lib
```

**Obtain the Utility Library**

CIAO’s CIDL compiler uses the Utility library. Download the Utility 1.2.2 library from the following location:

```bash
http://www.dre.vanderbilt.edu/cidlc/prerequisites/Utility-1.2.2.tar.bz2
```
There is nothing to build. The remaining instructions assume that the Utility library has been unzipped into a directory called C:\Utility-1.2.2.

### Set Up the Build Environment

Set CIAO_ROOT, XERCESCROOT, and UTILITY_ROOT environment variables and update your PATH. Setting CIAO_ROOT is not strictly necessary on Windows, but it makes using CIAO more convenient. Setting XERCESCROOT and UTILITY_ROOT is necessary.

For example:

```bash
set CIAO_ROOT=%TAO_ROOT%\CIAO
set XERCESCROOT=C:\xerces-c-src_2_6_0
set UTILITY_ROOT=C:\Utility-1.2.2
set PATH=%PATH%;%XERCESCROOT%\bin;%CIAO_ROOT%\bin
```

The %XERCESCROOT%\bin directory contains the Xerces DLLs. The %CIAO_ROOT%\bin directory contains the CIAO CIDL compiler.

Update the include and library directories in Visual Studio. Add the Boost root directory to Visual Studio’s include directories:

```
C:\Boost-1.30.2
```

Add the Boost lib directory that we created to Visual Studio’s library directories:

```
C:\Boost-1.30.2\lib
```

### Enable CIAO and the CIDL Compiler in MPC’s Default Features File

Create or edit the %ACE_ROOT%\bin\MakeProjectCreator\config\default.features file and enable CIAO and the CIDL compiler.

```
ciao = 1
cidl = 1
```

### Generate Build Files with MPC

Generate CIAO’s Visual Studio project files with MPC:

```
  cd %CIAO_ROOT%
```
perl %ACE_ROOT%/bin/mwc.pl -recurse -type vc71

This command generates a Visual Studio solution file for each MPC workspace file found in the build tree.

**Build CIAO’s CIDL Compiler**

Build CIAO’s CIDL compiler by using the Visual Studio workspace %CIAO_ROOT%/CIDLC/CIDLCL.sln.

You may use the **Batch Build** command in Visual Studio to build the CIDL compiler’s libraries and executables. Alternatively, you may find that it is easier to build the libraries and executables from the command line, as follows:

```
cd %CIAO_ROOT%/CIDLC
devenv CIDLC.sln /build debug
```

**Build CIAO’s Libraries and DAnCE Executables**

Build CIAO’s libraries and DAnCE executables by using the Visual Studio workspace %CIAO_ROOT%/CIAO.sln.

You may use the **Batch Build** command in Visual Studio to build the libraries and configurations in which you are interested. Alternatively, you may find that it is easier to build just the configurations in which you are interested from the command line. For example:

```
cd %CIAO_ROOT%
devenv CIAO.sln /build debug
```

**32.3.2 Building CIAO on UNIX with GNU Make and gcc**

CIAO may be built with gcc versions 3.3 and later.

**Obtain and Build the Xerces C++ Library**

The source code for Xerces C++ can be obtained from <http://xml.apache.org/xerces-c>. At publication time, the latest version of Xerces C++ is version 2.6. Download and unzip the Xerces C++ 2.6 source code. The remainder of this section assumes that Xerces C++ is installed in a directory called $HOME/xerces-c-src-2_6_0.
The Xerces C++ site also contains many prebuilt distributions of the Xerces C++ library. If you find a binary distribution that matches your platform and compiler then you can avoid building Xerces C++.

Set the XERCESCROOT environment variable to your root Xerces C++ directory, as follows:

```bash
export XERCESCROOT=$HOME/xerces-c-src_2_6_0
```

Build the Xerces C++ libraries as follows:

```bash
cd $XERCESCROOT/src/xercesc
autoconf
./runConfigure -plinux -cgcc -xg++ -minmem -nsocket -tnative -rpthread
make
```

This execution of the Xerces C++ runConfigure script command uses the gcc compiler, targets the linux platform, and builds with pthreads. For more information, type enter the following at the command line:

```bash
./runConfigure -help
```

The Xerces C++ project does not install its include files in the directories where CIAO is expecting them. We manually rectify this by creating a symbolic link to an include directory.

```bash
cd $XERCESCROOT
ln -s src include
```

**Obtain and Build the Boost library**

CIAO’s CIDL compiler uses the Boost regex and filesystem libraries and the spirit parser framework. The spirit parser framework consists only of header files.

CIAO’s UNIX versions can use the latest version of Boost, which is version 1.32 at publication time. Using a later version of Boost allows more flexibility in the choice of compiler version. The Boost source tree and the latest version of the Boost Jam build system can be downloaded from the Boost web site, <http://www.boost.org>.
Install Boost 1.32 and the latest Boost Jam in the directories of your choice. For this example, we assume that Boost 1.32 is installed in $HOME/boost_1_32_0 and Boost Jam is installed in $HOME/boost-jam.

Build the Boost 1.32 libraries with Boost Jam as follows:

```
cd $HOME/boost_1_32_0
$HOME/boost-jam/bjam -sTOOLS=gcc
```

Obtain the Utility Library
CIAO’s CIDL compiler uses the Utility library. Download the Utility 1.2.2 library from the following location:

http://www.dre.vanderbilt.edu/cidlc/prerequisites/Utility-1.2.2.tar.bz2

There is nothing to build. The remaining instructions assume that the Utility library has been unzipped into a directory called $HOME/Utility-1.2.2.

Set Up the Build Environment
Set CIAO_ROOT, XERESCRROOT, UTILITY_ROOT, BOOST_ROOT, BOOST_INCLUDE, and BOOST_LIB environment variables and update your PATH. Setting CIAO_ROOT is not strictly necessary on Windows, but it makes using CIAO more convenient. Setting XERESCRROOT, UTILITY_ROOT, and the Boost environment variables is necessary.

For example:

```
export CIAO_ROOT=$TAO_ROOT/CIAO
export XERESCRROOT=$HOME/xerces-c-src_2_6_0
export UTILITY_ROOT=$HOME/Utility-1.2.2
export BOOST_ROOT=$HOME/boost_1_32_0
export BOOST_INCLUDE=$BOOST_ROOT
export BOOST_LIB=$BOOST_ROOT/libs
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$XERESCRROOT/lib
```

Enable CIAO and the CIDL Compiler in MPC’s Default Features File
Create or edit the
$ACE_ROOT/bin/MakeProjectCreator/config/default.features file and enable CIAO and the CIDL compiler.

```
ciao = 1
cidl = 1
```
Generate Build Files with MPC
Generate CIAO’s GNU Makefiles with MPC:

```
cd $CIAO_ROOT
$ACE_ROOT/bin/mwc.pl -recurse -type gnuace
```

Build CIAO’s CIDL Compiler
Build CIAO’s CIDL compiler by executing `make` in the `$CIAO_ROOT/CIDLC` directory.

```
cd $CIAO_ROOT/CIDLC
make ciao=1 cidl=1
```

Build CIAO’s Libraries and DAnCE Executables
Build CIAO’s libraries and DAnCE executables by executing `make` in the `$CIAO_ROOT/ciao` and `$CIAO_ROOT/ciao` directories.

```
cd $CIAO_ROOT/ciao
make ciao=1 cidl=1
```

```
cd $CIAO_ROOT/DAnCE
make ciao=1 cidl=1
```

### 32.4 DAnCE Executable Reference

CIAO’s Deployment And Configuration Engine (DAnCE), which implements the OMG “Deployment and Configuration of Component-based Distributed Applications” specification (OMG Document ptc/03-07-08), contains a set of executables to dynamically load component libraries, create component instances, and make connections between them.

The DAnCE executables are described in the following subsections.

#### 32.4.1 Overview

The DAnCE executables are as follows:
UNIX and UNIX-like Systems

Table 32-11 DAnCE Executables

<table>
<thead>
<tr>
<th>Name</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeManager</td>
<td>$CIAO_ROOT/DAnCE/NodeManager/Node_Daemon</td>
</tr>
<tr>
<td>NodeApplication</td>
<td>$CIAO_ROOT/DAnCE/NodeApplication</td>
</tr>
<tr>
<td>ExecutionManager</td>
<td>$CIAO_ROOT/DAnCE/ExecutionManager</td>
</tr>
<tr>
<td>RepositoryManager</td>
<td>$CIAO_ROOT/DAnCE/RepositoryManager/executor</td>
</tr>
</tbody>
</table>

Windows Systems

Table 32-12 DAnCE Executables

<table>
<thead>
<tr>
<th>Name</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeManager</td>
<td>%CIAO_ROOT%\DAnCE\NodeManager\Node_Daemon.exe</td>
</tr>
<tr>
<td>NodeApplication</td>
<td>%CIAO_ROOT%\DAnCE\NodeApplication.exe</td>
</tr>
<tr>
<td>ExecutionManager</td>
<td>%CIAO_ROOT%\DAnCE\ExecutionManager.exe</td>
</tr>
<tr>
<td>RepositoryManager</td>
<td>%CIAO_ROOT%\DAnCE\RepositoryManager\executor.exe</td>
</tr>
</tbody>
</table>

32.4.2 Node Manager and Node Application

The NodeManager is a daemon process that launches NodeApplication processes as directed by the ExecutionManager. Each object reference in the Messenger’s ApplicationNodeMap.dat configuration file refers to a NodeManager object in the NodeManager daemon process. Recall that the Messenger’s ApplicationNodeMap.dat file is as follows:

```
Administrator_Node   corbaloc:iiop:localhost:10000/NodeManager
First_Receiver_Node  corbaloc:iiop:localhost:20000/NodeManager
Second_Receiver_Node corbaloc:iiop:localhost:30000/NodeManager
Messenger_Node       corbaloc:iiop:localhost:40000/NodeManager
```

The node map file expects to find four different NodeManager objects on the localhost, each in an ORB listening on a different port. Presumably, each NodeManager object lives in a different process. Each NodeManager launches a NodeApplication process as a container for the component instance or instances mapped to it. For example:

```
$CIAO_ROOT/DAnCE/NodeManager/Node_Daemon \ 
-ORBListenEndpoints iiop://localhost:10000 \ 
-s "$CIAO_ROOT/DAnCE/NodeApplication/NodeApplication"
```
The `NodeManager` executable recognizes the following command-line options:

### Table 32-13 NodeManager Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-s node_application_path</code></td>
<td>Path of the <code>NodeApplication</code> executable to be launched.</td>
<td>REQUIRED</td>
</tr>
<tr>
<td><code>-o ior_file</code></td>
<td>Export the <code>NodeManager</code> IOR to a file. Supersedes registration with Naming Service.</td>
<td>off</td>
</tr>
<tr>
<td><code>-d spawn_delay</code></td>
<td>Delay spawning of the <code>NodeApplication</code> by <code>spawn_delay</code> seconds. This can be helpful for debugging.</td>
<td>0</td>
</tr>
<tr>
<td><code>-n</code></td>
<td>Register the <code>NodeManager</code> with the Naming Service in the root naming context with the name retrieved by calling <code>ACE_OS::hostname()</code>. Superseded by export of IOR file.</td>
<td>off</td>
</tr>
<tr>
<td><code>-?</code></td>
<td>Display usage information.</td>
<td>n/a</td>
</tr>
</tbody>
</table>

#### 32.4.3 Execution Manager

The `ExecutionManager` reads the node map file and maps each component instance to the `NodeManager` responsible for it. For example:

```bash
$CIAO_ROOT/DAnCE/ExecutionManager/Execution_Manager \ 
-o em.ior -i ApplicationNodeMap.dat
```

The `ExecutionManager` executable recognizes the following command-line options:

### Table 32-14 Execution Manager Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-i node_map_file</code></td>
<td>Path of the node map file.</td>
<td>deployment.dat</td>
</tr>
<tr>
<td><code>-o ior_file</code></td>
<td>Export the <code>ExecutionManager</code> IOR to a file. Supersedes registration with Naming Service.</td>
<td>off</td>
</tr>
</tbody>
</table>
### 32.4.4 Repository Manager

The RepositoryManager parses the XML Deployment and Configuration files and passes the relevant deployment information to the ExecutionManager. For example:

```bash
$CIAO_ROOT/DAnCE/RepositoryManager/executor \
    -p package.tpd -d Application.cdp -k file://em.ior
```

The RepositoryManager executable recognizes the following command-line options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-k <code>execution_manager_ior</code></td>
<td>The Execution Manager’s IOR.</td>
<td>file://exec_mgr.ior</td>
</tr>
<tr>
<td>-p <code>package_url</code></td>
<td>The Top-level Package Descriptor’s URL REQUIRED</td>
<td></td>
</tr>
<tr>
<td>-d <code>plan_url</code></td>
<td>The Component Deployment Descriptor’s URL REQUIRED</td>
<td></td>
</tr>
</tbody>
</table>

### 32.5 CIDL Compiler Reference

A CIDL composition embedded in a CIDL file describes a component implementation. CIAO includes a CIDL compiler, cidlc, that generates local IDL interfaces for component homes and executors and C++ classes for servants and default executor implementations.

**Note**  
*The generated C++ code is only usable by CIAO. The C++ output from CIDL compilers cannot be interchanged among CORBA implementations. However,*
the code generated by CIAO’s CIDL compiler is platform-independent, making it possible to use CIAO in cross-compilation environments.

CIAO’s CIDL compiler maps CIDL files to equivalent IDL and C++ according to the CORBA Component Model specification.

### 32.5.1 CIDL Executables

**UNIX and UNIX-like Systems**
The CIDL compiler executable is `$CIAO_ROOT/bin/cidlc`

**Windows Systems**
The CIDL compiler executable is `%CIAO_ROOT%\bin\cidlc.exe`

**General Usage**
The general usage of the CIAO CIDL compiler is as follows:

```
cidlc <options> -- CIDL-file
```

The CIDL file name must be listed after the "--", which is listed after the options. For example:

```
cidlc -I . -I $CIAO_ROOT/ciao -I $TAO_ROOT \ 
-1 $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs -- Messenger.cidl
```

### 32.5.2 Output Files Generated

The CIDL compiler generates three files for each CIDL file. One of these files is an IDL2 file containing the component executor’s local IDL2 interfaces. The component developer compiles that file with the TAO IDL compiler. The remaining two files are C++ files containing the component servant’s class definition and implementation. The generation of these files ensures that the
generated code is portable and optimized for a wide variety of C++ compilers. The diagram illustrates the generated files.

Figure 32-34: Compiling a CIDL File

For a CIDL file named `Messenger.cidl`, running the command

```
cidl -I . -I $CIAO_ROOT/ciao -I $TAO_ROOT \
-I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs -- Messenger.cidl
```
generates the following files (we show how to customize these names later):

Table 32-16 IDL and C++ Files Generated

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessengerE.idl</td>
<td>IDL2 for the component executor, to be run through the tao_idl compiler.</td>
</tr>
<tr>
<td>Messenger_svnt.h</td>
<td>Component and facet servant class definition.</td>
</tr>
<tr>
<td>Messenger_svnt.cpp</td>
<td>Component and facet servant class implementations.</td>
</tr>
</tbody>
</table>

32.5.3 Using CIDL Compiler Options

We discuss CIDL compiler command line options in 32.5.4 through 32.5.9. To see a complete list of the CIDL compiler’s options, enter the following:

```
cidl --help
```

In addition to the CIDL compiler options listed by the `--help` argument, the CIDL compiler also recognizes the `-I` preprocessor argument for specifying an element of the include path.

32.5.4 Preprocessing Options

The CIDL compiler does not run the full C preprocessor. It recognizes only the `-I include_path` preprocessor command-line option and the `#include` preprocessor directive. All other preprocessor directives are ignored.

Each CIDL file must be compiled with, at a minimum, the following include path:

```
-I $CIAO_ROOT/ciao -I $TAO_ROOT -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs
```

Each CIDL file indirectly includes a standard IDL file called `Components.idl`, which in turn includes several other IDL files.

The table provides details of the preprocessing options.

Table 32-17 Preprocessing Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>---preprocess-only</td>
<td>Run the preprocessor on the IDL file, but do not generate any IDL or C++ code.</td>
<td>generate IDL and C++ code</td>
</tr>
</tbody>
</table>
### 32.5.5 General Options

The CIDL compiler has an option that allows you to turn on a verbose mode that displays detailed information about the CIDL file compilation steps. There are also two options for displaying usage information. The options are summarized in the table.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--trace-semantic-actions</td>
<td>Turn on verbose mode.</td>
<td>off</td>
</tr>
<tr>
<td>--help</td>
<td>Output usage information to stderr.</td>
<td>n/a</td>
</tr>
<tr>
<td>--help-html</td>
<td>Output usage information in HTML format to stderr.</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### 32.5.6 Servant File Options

The CIDL compiler generates a complete servant class implementation for each component and each facet. The component developer has some control over the servant’s usage of event type factories and the names of the generated files.

The table summarizes the servant-related CIDL compiler options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--suppress-register-factory</td>
<td>Suppress automatic registration of a value type factory for each event type. By default, a value type factory is automatically registered for each event type. If factory registration is suppressed, then the developer must manually register a value type factory for each event type.</td>
<td>off, meaning a value type factory is automatically registered for each event type</td>
</tr>
<tr>
<td>--svnt-hdr-file-suffix</td>
<td>Use this suffix instead of the default to construct the name of the servant’s header file.</td>
<td>_svnt.h</td>
</tr>
</tbody>
</table>
The CIDL compiler assumes that the servant is part of a dynamic library. On Windows platforms, classes exported from dynamic libraries must define an export macro. On UNIX-like platforms, the export macros define to nothing. However, future versions of the gcc compiler support the C++ export keyword, which may reduce code size by reducing the number of exported symbols. In either case, the export macros enable cross-platform development.

The CIDL compiler assumes that a component servant’s export macro is called <COMPONENT>_SVNT_Export and that the macro is defined in a header file called <Component>_svnt_export.h. For example, the Messenger component’s servant export macro is assumed to be

```
MESSENGER_SVNT_Export
```

and it is assumed to be defined in a C++ header file called

```
Messenger_svnt_export.h
```

If that is not the case, then use the --svnt-export-macro command-line argument to indicate the correct name of the export macro and the --svnt-export-include command-line argument to indicate the correct name of the export header file.

See 32.2.6, “Building the Messenger Application,” for more information on component export macros.
32.5.7 Local Executor File Options

The CIDL compiler generates an IDL file containing the component implementation’s local executor interfaces. The component developer implements the component and its facets by implementing these local executor interfaces.

The table summarizes the executor-related CIDL compiler options.

Table 32-20 Local Executor File Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--lem-file-suffix suffix</td>
<td>Suffix for the generated executor IDL file.</td>
<td>E</td>
</tr>
<tr>
<td>--lem-file-regex regex</td>
<td>Regular expression to use when constructing the name of the local executor IDL file.</td>
<td>n/a</td>
</tr>
<tr>
<td>--lem-force-all</td>
<td>Force generation of local executor mapping for all IDL types, whether used by the composition or not. By default, the CIDL compiler generates local executor interfaces only for those components used by the composition.</td>
<td>off</td>
</tr>
</tbody>
</table>

32.5.8 Starter Executor Implementation File Options

The CIDL compiler can generate a default executor implementation for each component and facet. These default executor implementation files contain empty C++ member function definitions that you fill in with your implementation code. This can be a great time saver.

Note: Running the CIDL compiler with the starter implementation options overwrites any existing implementation files of the same names. Any modifications will be lost unless you rename the starter implementation files after they are generated (recommended).

The table summarizes the implementation-related CIDL compiler options.

Table 32-21 Executor Implementation File Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--gen-exec-impl</td>
<td>Generate a default executor implementation class for each component and facet.</td>
<td>off</td>
</tr>
</tbody>
</table>
Table 32-21 Executor Implementation File Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--exec-hdr-file-suffix suffix</td>
<td>Use this suffix instead of the default to construct the name of the default executor implementation’s header file.</td>
<td>_exec.h</td>
</tr>
<tr>
<td>--exec-hdr-file-regex regex</td>
<td>Use this regular expression to construct the name of the default executor implementation’s header file.</td>
<td>n/a</td>
</tr>
<tr>
<td>--exec-src-file-suffix suffix</td>
<td>Use this suffix instead of the default to construct the name of the default executor implementation’s source file.</td>
<td>_exec.cpp</td>
</tr>
<tr>
<td>--exec-src-file-regex regex</td>
<td>Use this regular expression to construct the name of the default executor implementation’s source file.</td>
<td>n/a</td>
</tr>
<tr>
<td>--exec-export-macro macro</td>
<td>Replace the default executor implementation’s default export macro with this export macro.</td>
<td>see below</td>
</tr>
<tr>
<td>--exec-export-include file</td>
<td>Replace the default executor implementation’s default export include file this file.</td>
<td>see below</td>
</tr>
</tbody>
</table>

You are strongly advised to rename the generated default executor implementation files before modifying them. Otherwise, the CIDL compiler will likely overwrite your changes. For example, rename

Message_exec.h and Message_exec.cpp to Message_exec_i.h and Message_exec_i.cpp.

The CIDL compiler assumes that the executor implementation is part of a dynamic library. On Windows platforms, classes exported from a dynamic library must define an export macro.

The CIDL compiler assumes that a component executor’s export macro is called <COMPONENT>_EXEC_Export and that the macro is defined in a header file called <Component>_exec_export.h. For example, the Messenger component’s executor export macro is assumed to be

```c
MESSENGER_EXEC_Export
```

and it is assumed to be defined in a C++ header file called

```c
Message_exec_export.h
```
If that is not the case, then use the `--exec-export-macro` command-line argument to indicate the correct name of the export macro and the `--exec-export-include` command-line argument to indicate the correct name of the export header file.

### 32.5.9 Descriptor File Options

The CIDL compiler generates a CORBA Component Descriptor for each component. However, the generated descriptor file is not usable to deploy a CCM application using CIAO’s DAnCE facility. The generated descriptor file is formatted in accordance with the deprecated “Packaging and Deployment” chapter of the OMG CORBA Component Model specification (OMG Document formal/02-06-65) rather than the updated OMG “Deployment and Configuration of Component-based Distributed Applications” specification (OMG Document ptc/03-07-08). Thus, we ignore the generated CORBA Component Descriptor files in our deployment.

The table summarizes the descriptor-related CIDL compiler options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--desc-file-suffix suffix</code></td>
<td>Use this suffix instead of the default to construct the name of the descriptor file.</td>
<td><code>.ccd</code></td>
</tr>
<tr>
<td><code>--desc-file-regex regex</code></td>
<td>Use this regular expression to construct the name of the descriptor file</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### 32.6 IDL3-to-IDL2 Compiler Reference

CIAO includes an IDL3-to-IDL2 compiler that generates IDL2-compatible interfaces for ORB implementations that do not recognize IDL3 keywords such as “component” and “provides”. This enables a client developed with a non-CCM-aware ORB to communicate with a CCM component. For example, a Java client built with an ORB such as JacORB can use CIAO’s IDL3-to-IDL2 output files as its interface to the Messenger component. Simply compile the Messenger’s IDL3 files with CIAO’s IDL3-to-IDL2 compiler, and then compile the IDL2 output with JacORB’s IDL compiler.

**Note**  
The generated IDL2 code is usable by any ORB.
CIAO’s IDL3-to-IDL2 compiler maps IDL3 files to equivalent IDL2 according to the Equivalent IDL sections of the CORBA Component Model specification.

For an example of using the IDL3-to-IDL2 compiler, please see the Administrator_Client_IDL2.mpc project in the $TAO_ROOT/DevGuideExamples/CIAO/Messenger directory.

### 32.6.1 IDL3-to-IDL2 Source Code

The source code for the IDL3-to-IDL2 compiler is in the $CIAO_ROOT/tools/IDL3_to_IDL2 directory. Build the code in that directory to create the tao_idl3_to_idl2 executable.

### 32.6.2 IDL3-to-IDL2 Executable

**UNIX and UNIX-like Systems**

The IDL3-to-IDL2 compiler executable is $ACE_ROOT/bin/tao_idl3_to_idl2.

**Windows Systems**

The IDL3-to-IDL2 compiler executable is %ACE_ROOT%\bin\tao_idl3_to_idl2.exe.

**General Usage**

The general usage of the CIAO IDL3-to-IDL2 compiler is as follows:

```
tao_idl3_to_idl2 -I $CIAO_ROOT -I $CIAO_ROOT/ciao -I $TAO_ROOT -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs <options> <idl3 files>
```

For example:

```
tao_idl3_to_idl2 -I $CIAO_ROOT -I $CIAO_ROOT/ciao -I $TAO_ROOT -I $TAO_ROOT/tao -I $TAO_ROOT/orbsvcs -I . Messenger.idl
```

The lengthy include path is necessary to enable the IDL3-to-IDL2 compiler to find CIAO’s CCM-related IDL files.
32.6.3 Output Files Generated

The IDL3-to-IDL2 compiler generates one IDL2 output file for each input file. A developer typically compiles that output file with another ORB’s IDL compiler. The diagram illustrates the generated files.

![Diagram showing the process of compiling an IDL file with IDL3-to-IDL2]

For an IDL3 file named `Messenger.idl`, running the command

```
tao_idl3_to_idl2 -I $CIAO_ROOT -I $CIAO_ROOT/ciao -I $TAO_ROOT -I $TAO_ROOT/orbsvcs -I . Messenger.idl
```

Figure 32-35: Compiling an IDL File with IDL3-to-IDL2
generates the following file:

**Table 32-23 IDL and C++ Files Generated**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messenger_IDL2.idl</td>
<td>Equivalent IDL2 for Messenger’s IDL3 file, to be run through another ORB’s IDL compiler.</td>
</tr>
</tbody>
</table>

The generated **Messenger_IDL2.idl** file includes CIAO’s **Components.idl** file, which contains IDL2 CCM declarations. When compiling the **Messenger_IDL2.idl** file with a non-CCM-aware ORB’s IDL compiler, the **Components.idl** file and the files it includes must be in the include path of that ORB’s IDL compiler.

In our example, we compile the IDL3 file **Messenger.idl**, which follows:

```idl
// file Messenger.idl
#include <Components.idl>
#include <Runnable.idl>
#include <Publication.idl>
#include <Message.idl>
#include <History.idl>

cOMPONENT Messenger {
    attribute string subject;

    PROVIDES Runnable control;
    PROVIDES Publication content;

    Publishes Message message_publisher;
    PROVIDES History message_history;
};

home MessengerHome manages Messenger {};
```

Note that you must also compile the included IDL3 files **Runnable.idl**, **Publication.idl**, **Message.idl**, and **History.idl** with the IDL3-to-IDL2 compiler.

The compiler generates the IDL2 file **Messenger_IDL2.idl**:

```idl
// file Messenger_IDL2.idl
#include "Components.idl"
#include "Runnable_IDL2.idl"
#include "Publication_IDL2.idl"
#include "Message_IDL2.idl"
#include "History_IDL2.idl"
```
interface Messenger : Components::CCMObject
{
    attribute string subject;
    Runnable provide_control ();
    Publication provide_content ();
    History provide_message_history ();

    Components::Cookie subscribe_message_publisher (in MessageConsumer consumer)
        raises (Components::ExceededConnectionLimit);

    MessageConsumer unsubscribe_message_publisher (in Components::Cookie ck)
        raises (Components::InvalidConnection);
};

interface MessengerHomeExplicit : Components::CCMHome
{
};

interface MessengerHomeImplicit : Components::KeylessCCMHome
{
    Messenger create ()
        raises (Components::CreateFailure);
};

interface MessengerHome : MessengerHomeExplicit, MessengerHomeImplicit
{
};

32.6.4 IDL3-to-IDL2 Compiler Options
We discuss IDL3-to-IDL2 compiler command line options in 32.6.5 through 32.6.6. To see a complete list of the IDL3-to-IDL2 compiler’s options, enter the following:

    tao_idl3_to_idl2 -u

32.6.5 Preprocessing Options
The IDL3-to-IDL2 compiler uses the same preprocessor as the tao_idl compiler. For more information on the preprocessor options and directives, please see 5.5. The most commonly used of these options is the -I option, which specifies a directory for the include path. For example:
32.6.6 General Options

The IDL3-to-IDL2 compiler’s other remaining options are summarized below. Each option’s function is identical to its matching option of the IDL compiler.

Table 32-24 General IDL3-to-IDL2 Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o output-directory</td>
<td>Subdirectory in which to place the generated stub and skeleton files.</td>
<td>Current directory</td>
</tr>
<tr>
<td>-t dir</td>
<td>Directory used by the IDL compiler for temporary files.</td>
<td>In UNIX, uses the value of the TMPDIR environment variable, if set, or /tmp by default. In Windows, uses the value of the TMP environment variable, if set, or the WINNT environment variable, if set, or the TEMP environment variable, if set, or the WINNT directory (on NT).</td>
</tr>
<tr>
<td>-v</td>
<td>Verbose flag. IDL compiler will print progress messages after completing major phases.</td>
<td>No progress messages displayed.</td>
</tr>
<tr>
<td>-d</td>
<td>Print the Abstract Syntax Tree (AST) to stdout.</td>
<td>AST is not displayed.</td>
</tr>
<tr>
<td>-w</td>
<td>Suppress warnings.</td>
<td>All warnings displayed.</td>
</tr>
<tr>
<td>-V</td>
<td>Print version information for front end and back end.</td>
<td>No version information displayed.</td>
</tr>
<tr>
<td>-Cw</td>
<td>Output a warning if two identifiers in the same scope differ in spelling only by case.</td>
<td>Error output is default.</td>
</tr>
<tr>
<td>-Ce</td>
<td>Output an error if two identifiers in the same scope differ in spelling only by case.</td>
<td>Error output is default.</td>
</tr>
<tr>
<td>-g gperf-path</td>
<td>Specify a path for the gperf program</td>
<td>$ACE_ROOT/bin/gperf</td>
</tr>
</tbody>
</table>
32.7 Future Topics

Several CCM and CIAO-related topics are beyond the scope of this chapter. They include the following:

- Component navigation
- Keyed component homes
- Home finders
- Life cycle categories service, process, and entity
- The IDL3 supports keyword

In addition, there are several capabilities that are expected to be addressed in future versions of CIAO. These include the following:

- Static application deployment
- Deployment of real-time applications
- Container-managed persistent using the Persistent State Service (PSS) and Persistent State Definition Language (PSDL) (OMG Document formal/02-09-06)
- Integration with Enterprise Java Beans
- Using the Real-Time Event Service or OMG Notification Service as the event delivery infrastructure
- Quality-of-Service
Part 6

Appendices
Appendix A

Configuring ACE/TAO Builds

This appendix discusses the different mechanisms and options for configuring builds of ACE and TAO. It discusses how to specify MPC features, GNU Make build flags, and C++ macros as well as some of the options available from each mechanism.

A.1 MPC Features

As discussed in 4.3.2.3, MPC features define which areas of functionality are enabled and disabled. They also determine what targets the generated build files construct and the details of how those components are built. When running MPC with ACE and TAO (using $ACE_ROOT/bin/mwc.pl or $ACE_ROOT/bin/mpc.pl), feature-related files are kept in the $ACE_ROOT/bin/MakeProjectCreator/config directory. The global feature file, global.features, defines the default values for features of ACE and TAO. You can specify any build-specific feature settings in default.features, which overrides the default values from the global features file. Any features that are not specified in either file are assumed to be enabled.
For example, to tell ACE and TAO to generate build files that compile SSL-related code, including the ACE SSL library and TAO SSLIOP pluggable protocol, place the following in your `default.features` file:

```plaintext
ssl=1
```

When using GNU makefiles, you also need to set the SSL makefile build flag to make sure that the libraries are actually built.

The table below summarizes the features that can be specified in the `default.features` file. Note, that disabling a feature may also disable other features, tests, and examples that depend on this feature. To build as intended, most of these features must be combined with corresponding build flag discussed in the next section.

### Table 1-1 MPC features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ace_codecs</td>
<td>Enables building of the ACE encoding/decoding mechanism. Supports Base64 transfer encoding.</td>
<td>enabled</td>
</tr>
<tr>
<td>ace_filecache</td>
<td>Enables building of the ACE cached virtual file system.</td>
<td>enabled</td>
</tr>
<tr>
<td>ace_other</td>
<td>Enables building of the ACE naming service and NT service-related functionality.</td>
<td>enabled</td>
</tr>
<tr>
<td>ace_svcconf</td>
<td>Enables building of the ACE service configurator functionality.</td>
<td>enabled</td>
</tr>
<tr>
<td>ace_token</td>
<td>Enables building of the ACE token service.</td>
<td>enabled</td>
</tr>
<tr>
<td>ace_uuid</td>
<td>Enables building of the ACE UUID class.</td>
<td>enabled</td>
</tr>
<tr>
<td>acexml</td>
<td>Enables building ACE XML support. Required for TAO Implementation Repository, Notification Service persistence, and CIAO.</td>
<td>enabled</td>
</tr>
<tr>
<td>ami</td>
<td>Enables building TAO Asynchronous Method Invocation support.</td>
<td>enabled</td>
</tr>
<tr>
<td>boost</td>
<td>Specifies that the boost library is present. Required for building CIAO’s CIDL compiler.</td>
<td>disabled</td>
</tr>
<tr>
<td>ciao</td>
<td>Enables building of CIAO.</td>
<td>disabled</td>
</tr>
<tr>
<td>cidl</td>
<td>Enables building of CIAO’s CIDL compiler.</td>
<td>disabled</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>corba_messaging</td>
<td>Enables building of TAO’s CORBA messaging specification support. This includes AMI and the CORBA policy framework.</td>
<td>enabled</td>
</tr>
<tr>
<td>dummy</td>
<td>Dummy feature that prevents certain obsolete components from building. Should never be enabled.</td>
<td>disabled</td>
</tr>
<tr>
<td>dummy_label</td>
<td>Dummy feature that prevents certain obsolete components from building. Should never be enabled.</td>
<td>disabled</td>
</tr>
<tr>
<td>ec_TYPED_EVENTS</td>
<td>Enables support for typed event channels in the COS Event Service.</td>
<td>enabled</td>
</tr>
<tr>
<td>exceptions</td>
<td>Enables use of native C++ exceptions and the standard CORBA C++ error reporting mechanism.</td>
<td>enabled</td>
</tr>
<tr>
<td>Ft_reactor</td>
<td>Enable support for using a reactor that integrates event handling with the Fast Light Toolkit.</td>
<td>disabled</td>
</tr>
<tr>
<td>interceptors</td>
<td>Enables support for Portable Interceptors in TAO. Security, fault tolerance, and load balancing features all depend on interceptors.</td>
<td>enabled</td>
</tr>
<tr>
<td>java</td>
<td>Specifies that java SDK is present. Required for certain interoperability tests.</td>
<td>disabled</td>
</tr>
<tr>
<td>mfc</td>
<td>Configures ACE/TAO build for use with the Microsoft Foundation Class library. This feature is only for use with Microsoft Visual C++.</td>
<td>disabled</td>
</tr>
<tr>
<td>minimum_corba</td>
<td>Enabling this feature disables a number of services, tests, and examples that cannot work when TAO is built in a minimum CORBA configuration.</td>
<td>disabled</td>
</tr>
<tr>
<td>negotiate_codesets</td>
<td>Enabling this feature links TAO clients and servers with the TAO Codeset library, giving them the ability to negotiate codesets with other processes.</td>
<td>disabled</td>
</tr>
<tr>
<td>openssl</td>
<td>Specifies that the OpenSSL library is available.</td>
<td>enabled</td>
</tr>
<tr>
<td>qos</td>
<td>Enables build support for the ACE QoS library. Required for the TAO AV Streaming service.</td>
<td>disabled</td>
</tr>
<tr>
<td>qt</td>
<td>Enable support for using a reactor that integrates event handling with the Qt library.</td>
<td>disabled</td>
</tr>
<tr>
<td>repo</td>
<td>Enabling this feature disables some Load Balancing and IDL tests</td>
<td>disabled</td>
</tr>
<tr>
<td>rmcast</td>
<td>Enables build support for the ACE RMCast library.</td>
<td>enabled</td>
</tr>
<tr>
<td>rpc</td>
<td>Specifies that Remote Procedure Call (RPC) is present. Required for RPC-related performance tests.</td>
<td>disabled</td>
</tr>
</tbody>
</table>
When MPC generates GNU makefiles using the default `gnuace` project type, the makefiles support a number of build flags that are used to customize your build. Some of these are identically named as the MPC features discussed in the previous section and simply cause compilation of the projects and/or functionality that their corresponding feature caused to be included in the makefiles.

### A.2 GNU Make Build Flags

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rt_corba</code></td>
<td>Enables building of TAO’s Real-Time CORBA specification support.</td>
<td>enabled</td>
</tr>
<tr>
<td><code>rwho</code></td>
<td>Enables building of the Distributed rwho utility.</td>
<td>enabled</td>
</tr>
<tr>
<td><code>ssl</code></td>
<td>Specifies that Secure Sockets Layer (SSL) is present. Required for ACE SSL and TAO SSLIOP libraries.</td>
<td>disabled</td>
</tr>
<tr>
<td><code>threads</code></td>
<td>Enables multithreading support and components dependent on it.</td>
<td>enabled</td>
</tr>
<tr>
<td><code>uses_wchar</code></td>
<td>Enabling this feature disables various examples and utilities that do no support wide character builds.</td>
<td>disabled</td>
</tr>
<tr>
<td><code>wfmo</code></td>
<td>Disabling this feature disables examples that use the Wait For Multiple Objects (WFMO) reactor.</td>
<td>enabled</td>
</tr>
<tr>
<td><code>wince</code></td>
<td>Enables the Front-end for ACE CE (FaCE), a simple front-end framework for testing and debugging non-graphical application on Windows CE.</td>
<td>disabled</td>
</tr>
<tr>
<td><code>winregistry</code></td>
<td>Enables examples dependent on use of the Windows Registry.</td>
<td>enabled</td>
</tr>
<tr>
<td><code>wxWindows</code></td>
<td>Specifies that wxWindows library is present. Required for TAO’s wxNamingViewer utility and the ACE Configuration Viewer.</td>
<td>disabled</td>
</tr>
<tr>
<td><code>xt_reactor</code></td>
<td>Enable support for using a reactor that integrates event handling with the X Toolkit.</td>
<td>disabled</td>
</tr>
<tr>
<td><code>zlib</code></td>
<td>Specifies that the Zlib compression library is present and should be used.</td>
<td>disabled</td>
</tr>
<tr>
<td><code>zzip</code></td>
<td>Specifies that the ZZip compression library is present and should be used.</td>
<td>disabled</td>
</tr>
</tbody>
</table>
You can specify any of the GNU Make flags in the
$ACE_ROOT/include/makeinclude/platform_macros.GNU file or
using the MAKEFLAGS environment variable as well as on the command line
when you invoke make. Because many of the flags require that they be used
consistently across your application, the platform_macros.GNU file is
usually the preferred place to put these flags. When installing from the OCI
TAO CD, the installer places in your build tree the platform_macros.GNU
file that was used to generate the installed binaries.

When not using the ACE/TAO GNU makefile system to build your
application, it is your responsibility to ensure that compatible environment
variables and compiler options are used in building your application. Many of
the build flags effects can also be specified via preprocessor macro definitions
in the $ACE_ROOT/ace/config.h file.

The table below lists the GNU Make flags that affect not only the way the
ACE and TAO libraries and executables are built, but also the way your
applications are built. As indicated in the table’s right-most column, when you
build your applications, there are certain flags whose settings must match the
settings that were used when ACE and TAO were built. For example,
exceptions: If ACE and TAO were built to use native exceptions, then
applications must use native exceptions also. Other flags only affect the target
being built at that time. For example, debug: If ACE and TAO were built with
debug enabled, the applications do not have to be built with debug enabled,
and vice-versa.

Table 1-2 lists each GNU Make flag, a brief description of the flag, the default
setting of the flag (for most UNIX and UNIX-like platforms) for building
ACE and TAO, and the way applications must use the flag with respect to how
ACE and TAO were built. Almost all of the GNU Make flags are enabled by
setting them to one (1), and disabled by setting them to zero (0).

OCI’s recommended build flags are included in the release notes.

Table 1-2 GNU Make Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Default</th>
<th>Must Match?</th>
</tr>
</thead>
<tbody>
<tr>
<td>exceptions</td>
<td>Enable native C++ exceptions</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>include_env</td>
<td>Enable ACE_TRY_ENV macro backward compatibility mode</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>rtti</td>
<td>Enable C++ RTTI</td>
<td>1</td>
<td>N</td>
</tr>
</tbody>
</table>
### Table 1-2 GNU Make Flags

| Flag            | Description                                           | Default | Must Match?
|-----------------|-------------------------------------------------------|---------|----------------
| inline          | Enable C++ function inlining                          | 1       | Y              
| debug           | Enable support for debugging                          | 1       | N              
| optimize        | Turn on compiler optimization                        | 1       | N              
| repo            | Use GNU template repository                          | 0       | N              
| fast            | Use -fast compiler option (SunCC only)                | 0       | N              
| threads         | Enable threads (if OS supports)                       | 1       | Y              
| purify          | Third party product support                           | 0       | N              
| quantify        | Third party product support                           | 0       | N              
| shared_libs     | Build shared libraries                                | 1       | E              
| static_libs     | Build archive libraries                               | 0       | E              
| shared_libs_only| Build shared libraries only                           | 0       | E              
| static_libs_only| Build archive libraries only                          | 0       | E              
| minimum_corba   | Build with minimum CORBA support                      | 0       | Y              
| probe           | Enable ACE_Timeprobes                                 | 0       | B              
| profile         | Enable profiling                                      | 0       | N              
| xt_reactor      | Build with Xt reactor                                 | 0       | Y              
| fl_reactor      | Build with fl (Fast Light) reactor                    | 0       | Y              
| tk_reactor      | Build with tk reactor                                 | 0       | Y              
| qt_reactor      | Build with Qt reactor                                 | 0       | Y              
| gtk_reactor     | Build with GTK reactor                                | 0       | Y              
| ami             | Enable Asynchronous Method Invocation (AMI)           | 1       | Y              
| corba_messaging | Enable CORBA Messaging                                | 1       | Y              
| rt_corba        | Enable Real-time CORBA support                        | 1       | Y              
| interceptors    | Enable Portable Interceptors                          | 1       | Y              
| ssl             | Build with SSL support in ACE and TAO                 | 0       | B              
| stlport         | Build with STLport support                            | 0       | B              
| rwho            | Build the distributed rwho utility                    | 1       | B              

1. Must Match? signifies whether the flag must exactly match the value specified in the table.
### A.3 Using the Build Flags

In the remainder of this section, we describe each flag listed in A.2, including how to specify it using a GNU Make flag, a preprocessor macro, or both.

The flags are described below. *They are shown in the form necessary for changing the default.*

#### exceptions
Enable or disable support for native C++ exception handling.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>exceptions=0</td>
<td>#undef ACE_HAS_EXCEPTIONS</td>
</tr>
</tbody>
</table>

If native exception handling is not enabled, the alternate mapping for exception handling described in 6.4 must be used. You can use the ACE
exception handling macros described in 6.5 to make your code portable across both environments that use native C++ exceptions and those that do not.

**include_env**
Enable or disable backward compatibility mode for use of `ACE_TRY_ENV` macros.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include_env=1</code></td>
<td><code>#define ACE_ENV_BKWD_COMPAT</code></td>
</tr>
</tbody>
</table>

This flag can be used along with `exceptions=1` to retain backward compatibility for code that uses the old-style ACE exception macros for achieving portability between environments where native C++ exceptions are used and environments where the alternate mapping using `CORBA::Environment` parameters are used. See 6.5 for more information on the use of `ACE_TRY_ENV` and associated macros.

**rtti**
Enable or disable support for C++ Run-Time Type Identification (RTTI).

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtti=0</code></td>
<td><code>#define ACE_LACKS_RTTI</code></td>
</tr>
</tbody>
</table>

**inline**
Enable or disable C++ function inlining.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inline=0</code></td>
<td><code>#define ACE_NO_INLINE</code></td>
</tr>
</tbody>
</table>

If inlining is enabled when the ACE and TAO libraries are built, `inline` hints are passed to the compiler for many simple functions, and the resulting libraries may contain some inlined code. If inlining is disabled when the ACE and TAO libraries are built, no `inline` hints are passed to the compiler, and the libraries will contain independent function entries for these simple functions. If inlining is enabled when your application is built, inlined ACE and TAO functions may be inserted into your code.
A.3 Using the Build Flags

**debug**
Causes the compiler to generate debugging information (e.g., by passing the \(-g\) option for some compilers), and builds an in-memory database of ACE objects.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug=0</td>
<td>#define ACE_NDEBUG</td>
</tr>
</tbody>
</table>

**optimize**
Generate time-optimized code. Note that some compilers do not allow both debug and optimize to be specified at the same time.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimize=0</td>
<td>None</td>
</tr>
</tbody>
</table>

**repo**
Use the GNU template repository.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>repo=1</td>
<td>None</td>
</tr>
</tbody>
</table>

**fast**
Enable the \(-fast\) option. This only applies if you are using a Sun Workshop C++ compiler that implements this option.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast=1</td>
<td>None</td>
</tr>
</tbody>
</table>

Note that the **fast** flag is mutually exclusive of the **debug** flag.

**threads**
Build the ACE and TAO libraries to use threads (only applies if the operating system supports multithreaded programming). We recommend not modifying this flag from its default setting.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads=0</td>
<td>See text.</td>
</tr>
</tbody>
</table>

The **threads** flag can also be specified in the **config.h** file, but that is more complicated as there are several variables that must be set in a way that is specific to the operating system on which ACE and TAO are being built. You
are advised not to try to change these variables. Instead, you should use the GNU Make flag above.

**purify**
Process the object code so that memory integrity checking can be performed using IBM Rational PURIFY®. Using the `purify` flag causes the `purify` command to precede the linker command during the linking phase of building your application. The `purify` command must be found in your `PATH` environment variable.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>purify=1</code></td>
<td>None</td>
</tr>
</tbody>
</table>

**quantify**
Process the object code so that performance profiling can be obtained using IBM Rational QUANTIFY®. Using the `quantify` flag causes the `quantify` command to precede the linker command during the linking phase of building your application. The `quantify` command must be found in your `PATH` environment variable.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>quantify=1</code></td>
<td>None</td>
</tr>
</tbody>
</table>

**shared_libs**
Shared libraries are built by default. If the `shared_libs_only` flag is used, this flag is also set. If the `static_libs_only` flag is used, this flag is ignored.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shared_libs=0</code></td>
<td>None</td>
</tr>
</tbody>
</table>

**static_libs**
Static libraries are not built by default. If the `static_libs_only` flag is used, this flag is also set. If the `shared_libs_only` flag is used, this flag is ignored.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static_libs=1</code></td>
<td>None</td>
</tr>
</tbody>
</table>
A.3 Using the Build Flags

**shared_libs_only**
Pseudo-flag that prevents static libraries from being built whether or not the `static_libs` flag is used.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared_libs_only=1</td>
<td>None</td>
</tr>
</tbody>
</table>

**static_libs_only**
Pseudo-flag that prevents shared libraries from being built whether or not the `shared_libs` flag is used.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>static_libs_only=1</td>
<td>None</td>
</tr>
</tbody>
</table>

**minimum_corba**
Build a minimally-supported CORBA system.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimum_corba=1</td>
<td>#define TAO_HAS_MINIMUM_CORBA 1</td>
</tr>
</tbody>
</table>

**probe**
Build ACE with time probes.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe=1</td>
<td>#define ACE_COMPILE_TIMEPROBES</td>
</tr>
</tbody>
</table>

**profile**
Enable the use of the GNU `gprof` tool.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile=1</td>
<td>None</td>
</tr>
</tbody>
</table>

**xt_reactor**
Build support into ACE for the Xt reactor on platforms that support the X Window System.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>xt_reactor=1</td>
<td>#define ACE_HAS_XT</td>
</tr>
</tbody>
</table>

If this flag is defined via a preprocessor macro in `config.h`, you must also set the `CPPFLAGS`, `LIBS`, and `LDFLAGS` macros during the make, such that the
appropriate libraries (e.g., libXt, libX11) are available during the link stage. See 20.4 for more information on using the Xt reactor with TAO.

**fl_reactor**
Build into ACE support for the FLTK (The Fast Light Tool Kit) reactor.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>fl_reactor=1</td>
<td>#define ACE_HAS_FL</td>
</tr>
</tbody>
</table>

If this flag is defined via a preprocessor macro in `config.h`, you must also set the CPPFLAGS, LIBS, and LDFLAGS macros during the make such that the appropriate libraries (e.g., libfltk, libXext, libX11) are available during the link stage.

**tk_reactor**
Build support into ACE for the tk reactor.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>tk_reactor=1</td>
<td>#define ACE_HAS_TK</td>
</tr>
</tbody>
</table>

If this flag is defined via a preprocessor macro in `config.h`, you must also set the CPPFLAGS, LIBS, and LDFLAGS macros during the make such that the appropriate libraries (e.g., libtk, libtcl) are available at the link stage.

**qt_reactor**
Build into ACE support for the Qt reactor from Trolltech.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>qt_reactor=1</td>
<td>#define ACE_HAS_QT</td>
</tr>
</tbody>
</table>

If this flag is defined via a preprocessor macro in `config.h`, you must also set the CPPFLAGS, LIBS, and LDFLAGS macros during the make such that the appropriate libraries (e.g., libqt, libX11) are available at the link stage. See 20.3 for more information on using the Qt reactor with TAO.

**gtk_reactor**
Build into ACE support for the GIMP Toolkit (GTK) reactor.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtk_reactor=1</td>
<td>#define ACE_HAS_GTK</td>
</tr>
</tbody>
</table>
A.3 Using the Build Flags

If this flag is defined via a preprocessor macro in config.h, you must also set the CPPFLAGS, LIBS, and LDFLAGS macros during the make such that the appropriate libraries (e.g., libglib, libgtk, libX11) are available at the link stage.

**ami**
Enable Asynchronous Method Invocation (AMI).

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>ami=0</td>
<td>#define TAO_HAS_AMI 0</td>
</tr>
</tbody>
</table>

If this flag is defined via a preprocessor macro in config.h, you must also set the TAO_HAS_AMI_CALLBACK and TAO_HAS_AMI_POLLER macros with respect to your particular needs. See 7.2 for more information on using AMI.

**corba_messaging**
Build with full CORBA Messaging support.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>corba_messaging=0</td>
<td>#define TAO_HAS_CORBA_MESSAGING 0</td>
</tr>
</tbody>
</table>

This flag is defined by default, unless minimum_corba is enabled. See Chapter 7 for more information on CORBA Messaging.

**rt_corba**
Build with full Real-Time CORBA support.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt_corba=0</td>
<td>#define TAO_HAS_RT_CORBA 0</td>
</tr>
</tbody>
</table>

This flag is defined by default, unless minimum_corba is enabled or corba_messaging is disabled. See Chapter 9 for more information on Real-Time CORBA.

**interceptors**
Build with support for Portable Interceptors.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>interceptors=0</td>
<td>#define TAO_HAS_INTERCEPTORS 0</td>
</tr>
</tbody>
</table>

Enabling this flag builds in support for Portable Interceptors as defined in the CORBA specification. This flag is defined by default, unless
**Configuring ACE/TAO Builds**

minimum corba is enabled. For example, if ACE and TAO are built with this flag enabled, the types and interfaces defined as part of the PortableInterceptor module will be included in TAO. See $TAO_ROOT/tao/PortableInterceptor.pidl for the specific type and interface definitions that are included. See Chapter 10 for more information on Portable Interceptors.

**ssl**
Build with SSL support in ACE and TAO.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl=1</td>
<td>None</td>
</tr>
</tbody>
</table>

Enables building the SSL-related libraries in ACE and TAO. Requires $SSL_ROOT environment variable to point to the OpenSSL installation directory.

**stlport**
Build with STLport support.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>stlport=1</td>
<td>#define ACE_HAS_STLPORT 1</td>
</tr>
</tbody>
</table>

Allow components that use the STLport library to be built.

**rwho**
Build the distributed rwho utility.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>rwho=0</td>
<td>None</td>
</tr>
</tbody>
</table>

**pipes**
Add -pipe compiler option for pipe support.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipes=0</td>
<td>None</td>
</tr>
</tbody>
</table>

For use with gcc only, this option may provide increased compilation speed, at the cost of increased memory usage during compilation.
### split
Splits ACE source files before compilation.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>split=1</td>
<td>None</td>
</tr>
</tbody>
</table>

Splitting the ACE source files allows processes linked with static libraries to only include the parts of ACE they require and minimizes the memory footprint of the application.

### sctp
Build ACE SCTP and TAO SCIOP support.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>sctp=lksctp</td>
<td>#define ACE_HAS_SCTP 1</td>
</tr>
</tbody>
</table>

This flag should be set to name the SCTP implementation to use, either lksctp or openss7. Since both of these SCTP implementations are linux kernel options, the SCTP-related features are currently only supported on linux.

### versioned_so
Add versioning to library names.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>versioned_so=0</td>
<td>None</td>
</tr>
</tbody>
</table>

If set to zero, this flag does not append version information to the library name. If set to one, this flag causes each shared library to be appended with the contents of the SOVERSION variable. If SOVERSION is not set, then the version number defaults to .major.minor.beta for DOC group releases and .major.minor.patch_level for OCI versions.

### wfmo
Build with Wait For Multiple Objects reactor support.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfmo=1</td>
<td>None</td>
</tr>
</tbody>
</table>

The WFMO reactor is only supported on Microsoft Windows and is usually enabled there automatically.
**winregistry**

Build with windows registry support.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>winregistry=1</td>
<td>None</td>
</tr>
</tbody>
</table>

The `winregistry` option is only available on Windows and is usually enabled there automatically.

**templates**

Specifies type of template instantiation.

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>templates=used</td>
<td>None</td>
</tr>
</tbody>
</table>

This flag takes a value of either *automatic* (the default), *explicit*, or *used*. Explicit template instantiation is not currently supported in TAO 1.4a as all supported platforms are currently working with automatic. The used value is only for use with the OSF cxx C++ compiler.

**static_link**

Link only static libraries to executables (uses GNU linker’s `-static` flag).

<table>
<thead>
<tr>
<th>GNU Make Flag</th>
<th>Preprocessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>static_link=1</td>
<td>None</td>
</tr>
</tbody>
</table>

This flag is only supported when using the GNU linker.
Appendix B

Choosing How To Build ACE and TAO

There are several ways to build ACE and TAO, depending upon your operating system, compiler, and preferences. This appendix will help you choose the type of build you need to perform.

The instructions in this and subsequent appendices apply only if you want to build the ACE and TAO libraries and executables directly from the source code distribution. Alternatively, for many platforms, you can install pre-built binary versions of the libraries and executables directly from the OCI CD-ROMs. See 2.2 and the instructions that come with the CD for more information.

Building ACE and TAO on UNIX with GNU Make
For UNIX and UNIX-like platforms, it is possible to build TAO similarly to previous releases, using provided configuration files and GNU Makefiles generated by MPC. This method is described in Appendix C.

Building ACE and TAO on Windows with Visual C++
You can build ACE and TAO on Windows (NT, 2000, XP) using Visual C++. See Appendix D for details.
Building ACE and TAO on Windows with Borland C++ Builder
You can build ACE and TAO on Windows (NT, 2000, XP) using Borland C++. See Appendix E for details.

Using ACE and TAO with VxWorks
You can build and use ACE and TAO on the VxWorks operating system. See Appendix F for details.

Using ACE and TAO with LynxOS
You can build and use ACE and TAO on the LynxOS operating system. See Appendix G for details.
This appendix discusses how to build the ACE and TAO libraries and executables after installing the source code. Alternatively, you can install pre-built binary versions of the libraries and executables directly from the OCI CD-ROMs. See the instructions that come with the CD for more information.

C.1 Building ACE and TAO on a UNIX System

ACE and TAO can be used on many versions of UNIX and UNIX-like platforms, including Linux, Solaris, AIX, HP-UX, Tru64, IRIX, and others.

Before you build the ACE and TAO libraries, you should review C.2 to learn about reducing the memory footprint of the ACE and TAO libraries. The following steps must be performed in order to build TAO. They will be discussed at length in the pages that follow.

1. Create a build tree. (Optional, but recommended.)
2. Set environment variables.
3. Configure the source code for your platform.
4. Generate makefiles. (Optional, but recommended.)
5. Build ACE and TAO.
6. Verify the build.

C.1.1 Create a Build Tree
MPC includes a Perl script called `clone_build_tree.pl` that you can use to create multiple build trees from a single, shared source tree. The builds are a duplicate of `ACE_wrappers`, and are placed in `ACE_wrappers/build/name` where `name` is the name you supply to identify the build. Each build can use a different compiler and different settings.

The script must be run from the `ACE_wrappers` directory. You can name your build anything you like. For example:

```
MPC/clone_build_tree.pl default
```

will create an `ACE_wrappers/build/default` directory with a complete duplicate of the directory structure of `ACE_wrappers`, with symbolic links to source code and other necessary files. The build directory will not contain any makefiles as it is assumed you will use MPC to generate them.

If you are modifying ACE or TAO, and have added or removed files in the original tree, then you should run `clone_build_tree.pl` again without any arguments. This will update all existing builds.

You may choose not to create a build tree. In this case you can put your makefiles directly in `ACE_wrappers`.

C.1.2 Set Environment Variables
The environment variables for building ACE and TAO are `ACE_ROOT` and `TAO_ROOT`. Assuming you installed the source directories under `/usr/local`, and created a new build tree as outlined in C.1.1, the following sh commands set these environment variables.

```
ACE_ROOT=/usr/local/ACE_wrappers/build/default; export ACE_ROOT
TAO_ROOT=$ACE_ROOT/TAO; export TAO_ROOT
```

For the ACE and TAO executables to link, the ACE and TAO libraries must be in the library path. In TAO 1.4a, the output library path is easily configured using MPC. By default, libraries are placed in `$ACE_ROOT/lib`. The
C.1 Building ACE and TAO on a UNIX System

following sh command sets the search path for libraries, assuming MPC was used to generate your makefiles:

    LD_LIBRARY_PATH=$ACE_ROOT/lib:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH

---

**Note** You must define ACE_ROOT when building the ACE and TAO libraries. TAO_ROOT defaults to $ACE_ROOT/TAO. The same settings must also be defined to use ACE and TAO. See C.1.3 for information on setting the platform_macros.GNU file.

---

C.1.3 Configure the Source Code for Your Platform

To achieve ACE and TAO’s portability on a wide variety of platforms with a minimum impact to the source code, platform dependencies for the source code and the makefiles are constrained to three files.

- $ACE_ROOT/ace/config.h
- $ACE_ROOT/include/makeinclude/platform_macros.GNU
- $ACE_ROOT/bin/MakeProjectCreator/config/default.features

These files serve as documentation of which settings were used to build ACE and TAO, and also to ensure that your own projects use consistent settings where necessary.

The config.h file defines C++ preprocesser macros that control operating system and C++ compiler characteristics and system library coverage. Most of these macros are of the form ACE_HAS_feature or ACE_LACKS_feature. Currently, we recommend also putting any TAO_HAS_feature options in this file, though in the future those may be moved to another file.

Here are some examples of preprocesser macros that control how ACE and TAO are built:

    ACE_LACKS_SIGNED_CHAR
    ACE_HAS_IP_MULTICAST
    TAO_HAS_MINIMUM_CORBA

The platform_macros.GNU file defines macros used by GNU Make to build ACE and TAO and applications, such as exceptions, inline, debug, and optimize. For details about available options for use in platform_macros.GNU, see Chapter 2 and Appendix A.
Here are some examples of build flags that affect how ACE and TAO and applications are built:

```plaintext
debug=0
optimize=1
fast=1
inline=0
```

The `default.features` file is optional, and is used by MPC when generating makefiles. MPC will not generate makefiles that require a feature that is not enabled, and enabling a feature can change the contents of a makefile. By default, features such as `ssl`, `qt`, and `qos` are disabled. MPC will not generate makefiles for any projects such as TAO’s SSLIOP library or ACE’s QoS library that require these features. In the future many of the settings from `platform_macros.GNU` and `config.h` may migrate to the `default.features` file.

Here are some examples of settings that may be found in the `default.features` file:

```plaintext
ssl=1
qos=1
```

**Creating the Platform Configuration Files**

The three configuration files described above do not exist in the ACE and TAO source code distribution; you have to create them.

Your `config.h` file should `#include` a `config-*.h` file that is specific to your platform, such as `config-hpux11.00.h`, `config-linux.h`, or `config-sunos5.9.h`. Then, you can add your own configuration options before the `#include`.

As an example, we will create configuration files to build ACE and TAO for Solaris 9 using the Sun ONE Studio 8 Compiler. We also enable `qos` and `ssl` features.

To support Solaris 9, we create the following `$ACE_ROOT/ace/config.h` file:

```plaintext
// Add any configuration macros here.
#include "ace/config-sunos5.9.h"
```
Your `platform_macros.GNU` file should include the correct file for the platform and compiler, such as `platform_hpx_aCC.GNU`, `platform_linux.GNU`, or `platform_sunos5_sunc++.GNU`. Then, you can add your own configuration options before the `include`.

For example, to support the Sun ONE Studio 8 Compiler, we create the following `$ACE_ROOT/include/makeinclude/platform_macros.GNU` file:

```plaintext
# Add any configuration options such as debug, release, and exceptions here.
include $(ACE_ROOT)/include/makeinclude/platform_sunos5_sunc++.GNU
```

**Note** Several platform-specific `config*.h` and `platform*.GNU` files are provided as part of the ACE and TAO source code distribution. These cover all of the OCI supported platforms, as well as other platforms. If files are not included for a particular platform, operating system, and compiler combination in which you are interested, you may be able to create one by starting with one of the provided files and modifying it slightly. OCI can help you port ACE and TAO to a new platform.

Finally, we create our `$ACE_ROOT/bin/MakeProjectCreator/config/default.features` file:

```plaintext
ssl=1
qos=1
```

### C.1.4 Generate Makefiles

We are now ready to use the MPC tool to generate the makefiles for our build.

```
cd $ACE_ROOT
bin/mwc.pl -type gnuace -recurse
```

This command will generate makefiles for any `.mwc` workspace files found in our build tree, as well as generating makefiles for any `.mpc` project files. By generating makefiles for workspaces, we will be able to build related projects with a single command.
Note See Chapter 4 for more information on using MPC.

C.1.5 Build ACE and TAO
This step briefly describes how to build ACE and TAO. For more detailed instructions, see $ACE_ROOT/ACE-INSTALL.html and $TAO_ROOT/TAO-INSTALL.html.

The makefiles require version 3.79.1 or later of GNU Make. If you do not have GNU Make version 3.79.1, you can obtain it at no cost via <http://www.theaceorb.com/references/>.

Assuming MPC was used to generate makefiles as described above, it is possible to build a complete ACE and TAO development installation by running a single `make` command from $TAO_ROOT:

```
cd $TAO_ROOT
make
```

This will build everything you need to use TAO. It will not build any examples, tests, or performance tests.

Note You must use GNU Make to build ACE+TAO on UNIX and UNIX-like platforms. The make command on your system may actually invoke the native operating system’s make program instead of GNU Make. If you are not sure, type `make --version`. If the make command on your system is not GNU Make, check with your system administrator to see if GNU Make is available. On some systems, you may be able to use the gmake command.

C.1.6 Verify Your Build
Once you have a complete TAO build, you may want to run a few tests to verify that it is working correctly. The TAO source code distribution includes some basic tests and performance tests that you can use.

- `$TAO_ROOT/tests/Hello` contains a basic “Hello, world!” CORBA client and server application. If this test does not work, then your TAO build or your run time environment is seriously impaired.
- `$TAO_ROOT/DevGuideExamples` contains all of the source code for the examples in this Developer’s Guide, along with MPC files and run scripts.
C.2 Customizing ACE and TAO Builds

- `$TAO_ROOT/tests/Param_Test` contains an application that exercises the ORB’s basic parameter passing conventions for many OMG IDL data types.

- `$TAO_ROOT/performance-tests/Cubit/TAO/IDL_Cubit` contains tests for certain optimizations, such as collocation. You can also use the IDL Cubit server and client to test passing various command line options to applications.

See the README file in each of the above directories for more information on building and running these tests. Each of the above directories also contains MPC files to use for building the test, and a Perl script named `run_test.pl` for semi-automated running of the test.

C.2 Customizing ACE and TAO Builds

In certain situations (e.g., embedded environments) you may want to minimize the memory footprint required by the ACE and TAO libraries. There are several build parameters you can customize to help reduce the size and memory footprint of these libraries. You can also disable MPC features to prevent generation of makefiles for related projects.

Note To customize your build, you must set these parameters before building ACE and TAO.

C.2.1 Minimizing the Size of the TAO Library

TAO provides support for the `minimumCORBA` specification from the OMG. The `minimumCORBA` specification omits the following features from the CORBA specification:

- Dynamic Skeleton Interface (DSI)
- Dynamic Invocation Interface (DII)
- Dynamic Any
- Interceptors
- Interface Repository
- Advanced POA features
- CORBA/COM interworking

To select the minimumCORBA subset of TAO, you can define the TAO_HAS_MINIMUM_CORBA preprocessor macro with a value of 1 in $ACE_ROOT/ace/config.h or include minimum_corba=1 in platform_macros.GNU. You must also add minimum_corba=1 to your default.features file before generating your makefiles. Building only the minimumCORBA subset typically results in about a 26% reduction in the size of the TAO library ($ACE_ROOT/lib/libTAO.*). However, it may not be appropriate for your application due to the omission of the features listed above.

You can choose the minimumCORBA configuration of TAO, but retain the advanced POA features, such as Servant Managers, Default Servant, and Adapter Activators, by defining TAO_HAS_MINIMUM_POA = 0 in $ACE_ROOT/ace/config.h before building TAO.

Here is an example of a config.h file that could be used on Linux to build TAO with minimumCORBA support, yet retain the advanced POA features:

```c
#define TAO_HAS_MINIMUM_CORBA=1
#define TAO_HAS_MINIMUM_POA=0
#include "ace/config-linux.h"
```

### C.2.2 Selectively Building TAO Services

All TAO service libraries ($ACE_ROOT/lib/libTAO_*.* *) are built by default. These libraries include all of TAO’s currently-supported services (see Chapter 23 for more about TAO’s services). You can reduce the number of services that are built (and thereby reduce the overall build time and disk space consumption) by excluding unused services. In previous versions of TAO, this was controlled by setting the TAO_ORBSVCS variable to contain only the names of the services you want to build. This is no longer supported. However, you can change directory to $TAO_ROOT/orbsvcs/orbsvcs and build individual projects such as CosNaming or RTEvent to build only the services you want to use. Below is a list of the projects located in $TAO_ROOT/orbsvcs/orbsvcs.

AV
CosConcurrency
CosEvent
CosEvent_Serv
C.2 Customizing ACE and TAO Builds

CosEvent_Skel
CosLifeCycle
CosLoadBalancing
CosNaming
CosNaming_Serv
CosNaming_Skel
CosNotification
CosNotification_Persist
CosNotification_Serv
CosNotification_Skel
CosProperty
CosTime
CosTrading
CosTrading_Serv
CosTrading_Skel
DsEventLogAdmin
DsEventLogAdmin_Serv
DsEventLogAdmin_Skel
DsLogAdmin
DsLogAdmin_Serv
DsLogAdmin_Skel
DsNotifyLogAdmin
DsNotifyLogAdmin_Serv
DsNotifyLogAdmin_Skel
ETCL
FTORB_Utils
FTRT_ClientORB
FTRT_EventChannel
FT_ClientORB
FT_ServerORB
FaultTolerance
FtRtEvent
HTIOP
IFRService
PortableGroup
RTCORBAEvent
RTCosScheduling
RTEvent
RTEventLogAdmin
RTKokyuEvent
RTOLDEvent
RTSched
RTSchedEvent
RT_Notification
SSLIOP
Security
Svc_Utils
Another way to build a subset of TAO services is to create your own workspace for use with MPC. You could make a copy of the existing $TAO_ROOT/TAOACE.mwc workspace file and modify it to build only the services that you want. This approach has the added advantages of portability and repeatability.

The following is an example of a custom workspace (.mwc) file located in $TAO_ROOT that contains everything necessary to build the CosNaming and RTEvent libraries along with the Naming_Service executable.

```plaintext
// $TAO_ROOT/CosNaming_RTEvent.mwc
workspace {
  ../ace
  ../apps/gperf/src
  tao
  TAO_IDL
  orbsvcs/orbsvcs/Svc_Utils.mpc
  orbsvcs/orbsvcs/CosNaming.mpc
  orbsvcs/orbsvcs/RTEvent.mpc
  orbsvcs/Naming_Service
}
```
Appendix D

Building ACE and TAO Using Visual C++

This appendix discusses how and why you might want to build a custom version of the ACE and TAO libraries and executables from the source code for the Windows platform using Visual C++.

D.1 Why Build a Custom Version?

- Unicode should typically be enabled for new applications on Windows platforms. This is not yet fully supported by TAO, but ACE can take advantage of it.
- The C++ standard library that comes with Visual C++ 6 is quite dated. If you are using Visual C++ 6, you may wish to rebuild ACE/TAO to use STLport or another custom standard C++ library.
- You might wish to participate in the ACE and/or TAO open-source projects.
- ACE and TAO have many other custom build options, and Visual C++ supports many different options as well.
Building ACE and TAO with Visual C++ requires several steps:

1. Create a build tree. (Optional, but recommended.)
2. Set environment variables.
3. Configure the source code for your platform.
4. Generate Project Files. (Optional, but recommended)
5. Build ACE and TAO.
6. Verify the build.

D.2.1 Create a Build Tree

MPC includes a Perl script called `clone_build_tree.pl` that can create multiple build trees from a shared source tree. The builds are a duplicate of `ACE_wrappers`, and are placed in `ACE_wrappers/build/<name>`. Each build can use a different compiler and settings.

The script must be run from `ACE_wrappers`, and you can name the build anything you like. For example:

```
clone_build_tree.pl default
```

will create a directory named `ACE_wrappers/build/default` with a complete duplicate of the directory structure of `ACE_wrappers`, with links to
source code and other necessary files. The build directory will not contain any project files, as it is assumed you will use MPC to generate them.

If you are modifying ACE or TAO, then you should run `clone_build_tree.pl` again without any arguments. This will update all existing builds. Unlike UNIX and UNIX-like platforms, which support symbolic links, Windows uses hard links that can be broken by many tools. For example, editing a file with the Visual C++ 6 editor appears to actually remove the existing file and create a new one in its place. This is not a problem with Notepad or Visual C++ 7. When `clone_build_tree.pl` updates an existing build, it will overwrite existing files with the version from the build directory if it is newer, and the original will be renamed with the addition of a `.bak` extension. For example, if you edit `ACE_wrappers\build\default\VERSION`, then `clone_build_tree.pl` will rename `ACE_wrappers\VERSION` to `ACE_wrappers\VERSION.bak`, and copy the one from the build directory.

You may choose not to create a build tree. In this case, you can use the default projects, or generate new ones directly in `ACE_wrappers`.

### D.2.2 Set Up Your Environment

You should create `ACE_ROOT` and `TAO_ROOT` environment variables, and update your `PATH`. This is not strictly necessary on Windows, but it makes using TAO more convenient.

For example:

```bash
set ACE_ROOT=c:\ACE_wrappers\build\default
set TAO_ROOT=%ACE_ROOT%\TAO
set PATH=%PATH%;%ACE_ROOT%\bin;%ACE_ROOT%\lib
```

TAO 1.4a requires two entries in your `PATH`. The first is the location for generated executables, and the second is the location for DLLs.

### D.2.3 Configure the Source Code for Your Platform

If you installed the ACE and TAO source from the CD, OCI’s Distribution of TAO you may skip this step unless you want to rebuild using different settings.

Before you are able to build the libraries and executables, you must configure the source for the environment. For all Win32 platforms, create a new file called `%ACE_ROOT%\ace\config.h` and add the following lines:
Building ACE and TAO Using Visual C++

#define ACE_HAS_STANDARD_CPP_LIBRARY 1
#include "ace/config-win32.h"

You can modify this file to suit your custom build requirements. The above settings are those we used for the TAO 1.4a CD. Here are some additional settings you may need to add to your config.h file:

- **Windows 95/98/ME**
  - If you want to be fully compatible with these operating systems, you should insert the following lines at the top of the config.h file:

    #define ACE_HAS_WINNT4 0
    #define ACE_HAS_WINSOCK2 1

- **Open Socket Limit**
  - By default, Windows’ sockets library allows only 64 simultaneous open sockets. You can add the following definition to increase this limit, however ACE already increases it to 1024 by default:

    #define FD_SETSIZE 256

- **Auto_Ptr**
  - If you decide to use STLport with Visual C++ 6 (and we recommend that you do), then you may also want to add the following to your config.h:

    #include "ace/config-win32.h"
    #undef ACE_AUTO_PTR_LACKS_RESET

D.2.4 **Generate Makefiles**

We can now use MPC to generate the Makefiles for our build:

    cd %ACE_ROOT%
    mwc.pl -type vc71 -recurse

This command will generate an .sln file for any .mwc workspace file found in the build tree, as well as generate a .vcproj files for each .mpc project files. You could substitute other types as well, although you should probably create a separate build tree to avoid conflicts with output files. For Visual C++
6, use -type vc6; for Visual C++ 7, use -type vc7. Use -type nmake to generate nmake Makefiles that are compatible with all of the above build tools.

D.2 Building ACE and TAO

D.2.5 Build the Libraries

All TAO libraries and executables can be built from single Visual C++ workspace found in %TAO_ROOT%\TAOACE.sln. In Visual C++ 6, the workspace file name is TAOACE.dsw.

You may use the Batch Build command in Visual C++ to build the libraries and configurations in which you are interested. Alternatively, you may find that it is easier to build just the configurations in which you are interested from the command line, as follows:

```bash
devenv TAOACE.sln /build debug /out build.log
```

or

```bash
msdev TAOACE.dsw /Y3 /make "ALL - Win32 Debug" /OUT build.log
```

An easy way to build the most commonly used TAO projects is to build the Naming_Service project. Its dependencies build ACE, TAO, the TAO IDL compiler, and some other common projects. You can build this project from the command line as follows:

```bash
devenv TAOACE.sln /build debug /project Naming_Service /out build.log
```

or

```bash
msdev TAOACE.dsw /Y3 /make "Naming_Service - Win32 Debug" /OUT build.log
```

The advantage of using this method over using Batch Build is that you do not build libraries and executables that you do not need, and you do not need to know which projects are required to build a particular service.

D.2.6 Verify Your Build

Once you have a complete TAO build, you may want to run a few tests to verify that it is working correctly. The TAO source code distribution includes some basic tests and performance tests that you can use.
Building ACE and TAO Using Visual C++

- `%TAO_ROOT%\tests\Hello` contains a basic “Hello, world!” CORBA client and server application. If this test does not work, then your TAO build or your run time environment is seriously impaired.

- `%TAO_ROOT%\DevGuideExamples` contains all of the source code for the examples in this Developer’s Guide, along with MPC files and run scripts.

- `%TAO_ROOT%\tests\Param_Test` contains an application that exercises the ORB’s basic parameter passing conventions for several OMG IDL data types.

- `%TAO_ROOT%\performance-tests\Cubit\TAO\IDL_Cubit` contains tests for certain optimizations, such as collocation. You can also use the IDL Cubit server and client to test passing various command line options to applications.

See the README file in each of the above directories for more information on building and running these tests. Each of the above directories also contains MPC files to use for building the test, and a Perl script named `run_test.pl` for semi-automated running of the test.

D.3 Build Notes

- In TAO 1.4a, MPC is used to generate all projects. Using MPC ensures that all project types are kept up to date and makes it easier to add features, such as precompiled headers, to new projects. Using MPC also ensures consistent settings across all projects.

- If you receive errors while building the static libraries for TAO, it may be because the IDL generated files do not yet exist. The easiest way to generate these files is to just build twice. On the first pass, Visual C++ will run the custom build command on the IDL files, which generates code from the IDL files. On the second pass, Visual C++ will be able to build without errors. Do not set dependencies between the various static library projects; doing so has the unwanted side-effect of including the complete contents of the dependent library within the library for the project, resulting in some very large libraries.

- If you join the ranks of contributors to ACE and TAO, you will probably find yourself rebuilding certain files often. If this is the case, you may
want to enable incremental linking, and/or minimal build in the settings for the project you are working on. If you are changing header files frequently, you may want to remove them from the list of precompiled headers for that project; otherwise, any change will likely cause the entire project to have to be rebuilt.

- You can build any project or workspace using Visual C++ from the command line. This is most easily shown with a few examples:
  - To build the Debug configuration of all the projects in the TAOACE workspace, use the following command for Visual C++ 7 or 7.1:

    devenv TAOACE.sln /build Debug

    or, for Visual C++ 6:

    msdev TAOACE.dsw /Y3 /make "ALL - Win32 Debug"

  - To clean all of the projects you can use:

    devenv TAOACE.sln /clean

    or, for Visual C++ 6:

    msdev TAOACE.dsw /make "ALL" /clean

The /Y3 option above is an undocumented feature of Visual C++ 6 that causes the build time of each target and the total build time to be printed with the build’s output.

- When you use the clean feature of Visual C++, it leaves many temporary files behind. To make sure all the unneeded files are cleaned, you can search for the following wildcards using the Windows Search utility on the %ACE_ROOT% folder, then delete the found files:

  *.pch;*.lib;*.dll;*.sbr;*.plg;*.ncb;*.opt;*.clw;*.idb;*.obj;*.ilk;*.bsc;*.exp;*.pdb;*.suo

- You must restart Visual C++ after making changes to environment variables. You may run into this problem the first time you build TAOACE.sln on a machine if you had Visual C++ running before setting ACE_ROOT, TAO_ROOT, and PATH.
Appendix E

Building ACE and TAO with Borland C++

This appendix discusses how to build the ACE and TAO libraries and executables from the source code on Windows using Borland C++. By “Borland C++”, we mean the free Borland C++ compiler or a version of Borland C++ Builder. Version 5.5 of the free compiler and version 5 and 6 of the Borland C++ Builder environment are supported. Newer Borland tools should also be able to build ACE and TAO, but we do not test those tools.

E.1 Building ACE and TAO with Borland C++

MPC does not generate project files for the Borland C++ Builder (BCB) Interactive Development Environment (IDE). It does generate Makefiles for Borland C++ make. All files with the extension .bmak are Borland make compliant Makefiles.

Building ACE and TAO with Borland C++ requires several steps:

1. Create a build tree. (Optional)
2. Set up your environment
3. Configure the source code for your platform
4. Generate Makefiles (Optional)
5. Build the libraries
6. Verify your build

The details of each step are discussed below

**E.1.1 Create a Build Tree**

*Note* *This feature requires use of NTFS, because it relies on linked files.*

MPC includes a Perl script called `clone_build_tree.pl` that can create multiple build trees from a shared source tree. The builds are a duplicate of `ACE_wrappers`, and are placed in `ACE_wrappers/build/<name>`. Each build can use a different compiler and settings.

The script must be run from `ACE_wrappers`, and you can name the build anything you like. For example:

```
clone_build_tree.pl default
```

will create an `ACE_wrappers/build/default` directory with a complete duplicate of the directory structure of `ACE_wrappers` with links to source code and other necessary files. The `build` directory does not contain any Borland Makefiles, as it is assumed you will use MPC to generate them.

When `clone_build_tree.pl` updates an existing build, it overwrites existing files with the version from the build directory if it is newer, and the original will be renamed with the addition of a `.bak` extension. For example, if you edit `ACE_wrappers\build\default\VERSION`, then `clone_build_tree.pl` will rename `ACE_wrappers\VERSION` to `ACE_wrappers\VERSION.bak`, and link the one from the build directory.

You may choose not to create a build tree. In this case, you can generate makefiles directly in `ACE_wrappers`.

**E.1.2 Set up your Environment**

You should create `ACE_ROOT` and `TAO_ROOT` environment variables, and update your `PATH`. This is not strictly necessary on Windows, but it makes using TAO more convenient.
E.1 Building ACE and TAO with Borland C++

For example:

```
set ACE_ROOT=c:\ACE_wrappers\build\default
set TAO_ROOT=%ACE_ROOT%\TAO
set PATH=%PATH%;%ACE_ROOT%\bin;%ACE_ROOT%\lib
```

TAO 1.4a requires two entries in your PATH. The first is the location for generated executables, and the second is the location for DLLs.

The Borland C++ environment also uses some special environment settings:

- **DEBUG** — Set to 1 if you want to build a debug version. This is the default.
- **RELEASE** — Set to 1 if you want to build a release version.
- **UNICODE** — Define this environment variable if you want to build a unicode version.

**Note** In order to build static versions of ACE and TAO, you must pass the -static option to MPC when you generate your makefiles.

E.1.3 Configure the Source Code for Your Platform

Before you can build the libraries and executables, you must configure the source code for your environment. For all Win32 platforms, create a new file called `%ACE_ROOT%\ace\config.h` and add the following line:

```
#include <ace/config-win32.h>
```

If you use this method, `config-win32.h` will automatically include the file `%ACE_ROOT%\ace\config-win32-borland.h`. You can add additional modifiers before the platform-specific configuration file. Here are some additional settings you may need to add to your `config.h` file:

- **Windows 95/98/ME**
  - If you want to be fully compatible with these operating systems, you should insert the following lines at the top of the `config.h` file:
    ```
    #define ACE_HAS_WINNT4 0
    #define ACE_HAS_WINSOCK2 1
    ```

- **Open Socket Limit**
- By default, Windows’ sockets library allows only 64 simultaneous open sockets. You can add the following definition to increase this limit, however ACE already increases it to 1024 by default:

```
#define FD_SETSIZE 1024
```

- Using Standard C++

If you want to use the Standard C++ headers that come with Borland C++ Builder 5.0 (e.g., iostream, cstdio, etc., as defined by the C++ Standard ISO IEC14882-1998(E)), add the following line to the top of your config.h file:

```
#define ACE_HAS_STANDARD_CPP_LIBRARY 1
```

E.1.4 Generate the Makefiles

The Borland makefiles can be generated by specifying the `bmake` project type.

```
cd %ACE_ROOT%
bin\mwc.pl -type bmake -recurse
```

If you are building static libraries, you must also pass the `-static` option when invoking `mwc.pl`.

E.1.5 Build the Libraries

Using Borland C++ `make`, you will build TAO and ACE from the command line. Change to the `%TAO_ROOT%` directory and type the following command to build TAO and ACE:

```
make
```

The type of configuration built is completely dependent upon your environment settings. The files that are built are stored in subdirectories of the location of the Makefile. The name of the subdirectory depends upon your environment settings, as follows:

- `DEBUG` or `RELEASE`, depending upon whether you are building a debug or a release version.
- `UNICODE`, when you defined the environment variable `UNICODE`. 
A suffix is appended to the names of the generated libraries and DLLs. A \texttt{d} is appended for a debug build, and an \texttt{s} is appended for a static build.

For example, the ACE DLL will be named \texttt{acesd.lib} for the Borland static debug build.

**E.1.6 Verify Your Build**

Once you have a complete TAO build, you may want to run a few tests to verify that it is working correctly. The TAO source code distribution includes some basic tests and performance tests that you can use.

- \%TAO\_ROOT\%\tests\Hello contains a basic “Hello, world!” CORBA client and server application. If this test does not work, then your TAO build or your run time environment is seriously impaired.

- \%TAO\_ROOT\%\DevGuideExamples contains all of the source code for the examples in this Developer’s Guide, along with MPC files and run scripts.

- \%TAO\_ROOT\%\tests\Param\_Test\ contains an application that exercises the ORB’s basic parameter passing conventions for several OMG IDL data types.

- \%TAO\_ROOT\%\performance\_tests\Cubit\TAO\IDL\_Cubit\ contains tests for certain optimizations, such as collocation. You can also use the IDL Cubit server and client to test passing various command line options to applications.

See the \texttt{README} file in each of the above directories for more information on building and running these tests. Each of the above directories also contains MPC files to use for building the test, and a Perl script named \texttt{run\_test.pl} for semi-automated running of the test.
Appendix F

Using ACE and TAO with VxWorks

This appendix describes how to build and use ACE and TAO with the VxWorks (version 5.5.x) operating system and the Tornado 2.2.x development environment from Wind River Systems, Inc. VxWorks is often used in embedded and real-time systems.

Cross-compilation is the only way to build ACE and TAO for VxWorks. Here, we will describe building ACE and TAO on a Windows host and on a UNIX host, such as Solaris, for a PowerPC target.

F.1 Kernel and System Configuration

The VxWorks kernel is highly configurable and may require some modification for optimal performance with ACE and TAO. The following values can be enabled in the kernel config.h (located in the bsp directory):

```c
#define INCLUDE_CPLUS_IOSTREAMS_FULL /* include all of iostreams */
#define INCLUDE_POSIX_ALL             /* include all available POSIX functions */
#define INCLUDE_LOADER                /* object module loading */
#define INCLUDE_SHELL                 /* interactive c-expression interpreter */
#define INCLUDE_ROUTE_SOCK            /* include routing socket interface */
```
For ACE and TAO to run properly under VxWorks, the macro `NUM_FILES` (defined in `configAll.h`) must be changed. This macro (defaulted to 50) must be changed to at least 64. The value increase is required due to the possible high number sockets that can be created under normal operation of TAO.

Optionally, if DNS is enabled (i.e., `INCLUDE_DNS_RESOLVER` is defined in your kernel `config.h`), you must modify the `RESOLVER_DOMAIN_SERVER` and `RESOLVER_DOMAIN` in your `configAll.h` (located in the file `$WIND_BASE/target/config/all`) to reflect your DNS information. If DNS is not enabled the command line argument `-ORB_DottedDecimalAddresses 1` must be used with TAO clients and servers to tell the ORB not to use host names under the IIOP protocol. Alternatively, `TAO_USE_DOTTED_DECIMAL_ADDRESSES` can be defined when compiling TAO to accomplish the same thing without requiring the command line option. (See F.2.)

Multicast routing is only available in VxWorks 5.4 or later. If you are planning to use multicast routing, you need to make the routing entry before running any TAO modules. This can be done at the VxWorks shell by calling `routeAdd()`. For example:

```
routenAdd("224.0.0.0", "192.168.133.5");
```

The second argument shown is an example IP address; replace it with the correct IP address or host name of your target.

Alternatively, you can add the following code to `main()` in your ACE and TAO programs if you do not want to manually run the `routeAdd()` command:

```
#include <routeLib.h>
#include <ace/OS_NS_unistd.h>
#include <ace/OS_NS_stdio.h>

char host[MAXHOSTNAMELEN];
if (ACE_OS::hostname(host, MAXHOSTNAMELEN) != -1) {
    if (routeAdd("224.0.0.0", host) != 0) {
```
ACE_OS::perror("Unable to add the multicast routing entry");
}
}
else {
    ACE_OS::perror("Unable to get the host name");
}

F.2  Environment Setup

ACE, gperf and tao_idl must be statically built for your host prior to setting up the environment to build for VxWorks. We describe how to do this for both UNIX (in F.2.1) and Windows (in F.2.2) build hosts.

---

**Note**  
*It is possible to build a shared version of ACE, gperf and tao_idl, but statically linking can help you avoid problems with library mismatches and finding the shared library at run time.*

---

F.2.1  UNIX

If your host is Solaris, Linux, or HP-UX, follow the directions laid out in Appendix C until you reach C.1.3, “Configure the Source Code for Your Platform”.

Next, create the following files for your host machine:

```
$ACE_ROOT/ace/config.h
$ACE_ROOT/include/makeinclude/platform_macros.GNU
```

For example, if your build host machine is running Solaris, and you are using the GNU compiler, you would create `platform_macros.GNU` in `$ACE_ROOT/include/makeinclude` with the following contents:

```
depth=0
optimize=1
static_libs_only=1
include $(ACE_ROOT)/include/makeinclude/platform_sunos5_g++.GNU
```

Your `config.h` file can be set up once and can take into account both the host and target platforms based on preprocessor macros. The following `config.h` file will suffice for Solaris, Linux, HP-UX, and Windows hosts, as well as cross-compiling for VxWorks:
Once the above configuration is complete, build a static version of ACE, gperf, and tao_idl. To do this, first generate a minimal set of makefiles with MPC.

```
cd $ACE_ROOT/TAO
../bin/mwc.pl -type gnuace TAOACE.mwc
```

Now, build the ACE library by running GNU Make in $ACE_ROOT/ace with the ACE target:

```
cd $ACE_ROOT/ace
make ACE
```

The gperf and tao_idl executables can be built similarly by running GNU Make with no parameters in the $ACE_ROOT/apps/gperf/src and $ACE_ROOT/TAO/TAO_IDL directories.

```
cd $ACE_ROOT/apps/gperf/src
make

cd $ACE_ROOT/TAO/TAO_IDL
make
```

Now, you can reconfigure your source tree to build for your target. First, clean up libraries and object files left from the static build for your host:

```
cd $ACE_ROOT/ace
```
F.2 Environment Setup

Next, modify your platform_macros.GNU file in $ACE_ROOT/include/makeinclude as follows:

```
TOOL=gnu
debug=0
optimize=1
static_libs_only=1
include $(ACE_ROOT)/include/makeinclude/platform_vxworks5.5.x.GNU
```

Next, set the Tornado related environment variables. These variables are required to build ACE and TAO:

```
WIND_BASE <tornado install directory>
WIND_HOST_TYPE <host type (e.g., sun4-solaris2, x86-win32,...)>
PATH $PATH:$WIND_BASE/host/$WIND_HOST_TYPE/bin
LD_LIBRARY_PATH $LD_LIBRARY_PATH:$WIND_BASE/host/$WIND_HOST_TYPE/lib
```

If your host platform is HP-UX, you may need to replace LD_LIBRARY_PATH with SHLIB_PATH in the above set of variables.

At this point, you can follow the instructions in C.1.4, “Generate Makefiles” and finish with the remaining instructions in Appendix C.

**Note**
You may run into difficulties building ACE and TAO with optimization turned on. The gcc-2.96 (2.96+) 19990621 Altivec VxWorks 5.5 compiler may produce an internal compiler error while compiling TAO. If this is the case, set optimize=0 in your platform_macros.GNU file and rebuild.

**Note**
The version of GNU Make (3.74) that comes with Tornado 2 will not function correctly when building ACE and TAO. Use version 3.79.1 or higher. This should not be a concern if the PATH environment variable is set as shown above.

F.2.2 Windows

If your host is Windows, follow the directions in Appendix D until you reach D.2.3, “Configure the Source Code for Your Platform”.

Next, create the following file for your host machine:
Using ACE and TAO with VxWorks

%ACE_ROOT%\ace\config.h

Your config.h file can be set up once and can take into account both the host and target platforms based on preprocessor macros. The following config.h file will suffice for Solaris, Linux, HP-UX, and Windows hosts, as well as cross-compiling for VxWorks:

```c
#if defined (sun)
# include "ace/config-sunos5.7.h"
#endif

#else if defined (linux)
# include "ace/config-linux.h"
#endif

#else if defined (__hpux)
# include "ace/config-hpux-11.00.h"
#endif

#else if defined (_MSC_VER) || defined (__BORLANDC__) || defined (_MINGW32_)
# define ACE_DISABLE_WIN32_ERROR_WINDOWS
# define ACE_HAS_STANDARD_CPP_LIBRARY 1
# define ACE_DISABLE_WIN32_INCREASE_PRIORITY
# include "ace/config-win32.h"
#endif

#else if defined (ACE_VXWORKS)
# include "ace/config-vxworks5.x.h"
#endif
```

Once the above configuration is complete, build a static version of ACE, gperf and tao_idl. To do this, first generate project and workspace/solution files using MPC.

For example, if you are using Visual C++ version 7.1:

```bash
cd %ACE_ROOT%\TAO
..\bin\mwc.pl -static -type vc71 TAOACE.mwc
```

For Visual C++ version 6.0, pass vc6 to the -type option.

Next, build the ACE library, gperf and tao_idl by opening the TAOACE workspace or solution file (e.g., TAOACE.sln) in the %ACE_ROOT%\TAO directory (with Visual C++), selecting the active build configuration of “Win32 Release,” and building the TAO_IDL_EXE project.

Next, create a platform_macros.GNU file in

%ACE_ROOT%\include\makeinclude with the following contents:

```makefile
TOOL=gnu
```
F.2 Environment Setup

debug=0
optimize=1
static_libs_only=1
include $(ACE_ROOT)/include/makeinclude/platform.vxworks5.5.x.GNU

Next, set the Tornado related environment variables. These variables are required to build ACE and TAO. If you are using the Cygwin bash shell, set the following environment variables:

WIND_BASE       <tornado install directory>
WIND_HOST_TYPE  <host type (e.g., sun4-solaris2, x86-win32,...)>
PATH            $PATH:$WIND_BASE/host/$WIND_HOST_TYPE/bin
LD_LIBRARY_PATH $LD_LIBRARY_PATH:$WIND_BASE/host/$WIND_HOST_TYPE/lib

If you are not using the Cygwin bash shell, set the WIND_BASE and WIND_HOST_TYPE variables, then set the PATH variable as shown below. You do not need to set LD_LIBRARY_PATH at all.

PATH            %PATH%;%WIND_BASE%;%WIND_HOST_TYPE%\bin

At this point, you can follow the directions in C.1.4, “Generate Makefiles” and finish with the remaining instructions in Appendix C.

---

**Note**  You may run into difficulties building ACE and TAO with optimization turned on. The gcc-2.96 (2.96+) 19990621 Altivec VxWorks 5.5 compiler may produce an internal compiler error while compiling TAO. If this is the case, you will need to set optimize=0 in your platform_macros.GNU file.

---

**Note**  The version of GNU Make (3.74) that comes with Tornado 2 will not function correctly when building ACE and TAO. Use version 3.79.1 or higher. This should not be a concern if the PATH environment variable is set as shown above and you are using the Cygwin bash shell.

---

If you are building on a Windows host and you are not using the Cygwin bash shell, you will need to install a few UNIX utilities (make, xargs, pwd, mkdir). You can download these utilities from <http://downloads.ociweb.com/unsupported/>. You will also need to add the following to your platform_macros.GNU file to avoid using the built-in mkdir of the Windows command prompt.
ACE_MKDIR = <full path to mkdir> -p
This appendix describes how to build and use ACE and TAO with the LynxOS (Version 3.1.0) operating system from LynuxWorks, Inc. ACE and TAO can be built with both native and cross-compilers for LynxOS. The approach you use depends mostly upon the target machine. If it does not have sufficient memory, disk space, or CPU speed, you may want to cross-compile on a host with more resources.

G.1 Native Compilation

G.1.1 Kernel and System Configuration
It may be necessary to make a few modifications to the system to build TAO on a machine running LynxOS. You may need super-user privileges to make these changes.

- You will need at least 256 MB of swap to build all of orbsvcs. Some simple modifications can be made if you do not have enough swap. If /.swap already exists, you will have to remove it before executing the following commands:
Using ACE and TAO with LynxOS

mkcontig /.swap 256m
prio 25 vmstart /.swap

This will create a contiguous file called .swap in the root directory. The swap space is then enabled at a priority of 25. This means that any process with a priority of 25 or lower can be swapped out at any time.

- You may need to increase the per process data and stack limits, as well. We use values of 655360 for the data limit and 8192 for the stack limit. These values are located in /etc/starttab. Modifications to this file require a reboot for the new values to take effect.

- The number of allowed open file descriptors may need to be changed to build orbsvcs. The default limit (defined in /usr/include/param.h) is 40 if your LynxOS installation is not a development installation. Since this is a kernel parameter, a new kernel must be generated to change the value. The TAO IDL compiler requires a higher number of file descriptors (at least 64) to process some files under orbsvcs. Please refer to the LynxOS manual for instructions on how to change kernel parameters.

- A run-time modification may be required if you wish to use multicast routing with TAO. If the multicast routing entry does not exist, then you must add one. Running netstat -r will show output similar to the following:

Routing tables
Internet:
<table>
<thead>
<tr>
<th>Destination</th>
<th>Gateway</th>
<th>Flags</th>
<th>Refs</th>
<th>Use</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>localhost</td>
<td>localhost</td>
<td>UH</td>
<td>1</td>
<td>32</td>
<td>lo0</td>
</tr>
<tr>
<td>192.168.0.0</td>
<td>link#10</td>
<td>UC</td>
<td>0</td>
<td>0</td>
<td>et0</td>
</tr>
<tr>
<td>krusty.ociweb.com</td>
<td>80:0:20:91:e5:a3</td>
<td>UHL</td>
<td>2</td>
<td>29</td>
<td>et0</td>
</tr>
<tr>
<td>mbx54.ociweb.com</td>
<td>80:0:3e:28:c5:80</td>
<td>UHL</td>
<td>1</td>
<td>4</td>
<td>lo0</td>
</tr>
<tr>
<td>224</td>
<td>mbx54.ociweb.com</td>
<td>UGS</td>
<td>0</td>
<td>0</td>
<td>et0</td>
</tr>
</tbody>
</table>

This routing table has a multicast routing entry. If yours does not have an entry similar to the one that starts with 224, then the following command will add the multicast routing entry:

    route add 224.0.0.0 'uname -n'
G.2 Cross-Compilation

G.2.1 Configuring and Building the Source Code
Setting up for cross-compilation is slightly different than configuring for
native compilation. The main difference is that you must statically build ACE,
gperf and tao_idl for your host prior to configuring to build for your target.

Note It is possible to build a shared version of ACE, gperf and tao_idl, but
statically linking can help you avoid problems with library mismatches and
with finding the shared library at run time.

Follow the directions in Appendix C until you reach C.1.3, “Configure the
Source Code for Your Platform”.

At this point, create the following files for your host machine:

$ACE_ROOT/ace/config.h
For example, if your build host machine is running Solaris, and you are using the GNU compiler, you would create platform_macros.GNU in $ACE_ROOT/include/makeinclude with the following contents:

```
debug=0
optimize=1
static_libs_only=1
include $(ACE_ROOT)/include/makeinclude/platform_sunos5_g++.GNU
```

Your $ACE_ROOT/ace/config.h file can be set up just once for all platforms using preprocessor macros. The following config.h file is sufficient for Solaris, Linux, and Windows hosts, as well as for cross-compiling for LynxOS:

```
#if defined (sun)
 # include "ace/config-sunos5.7.h"

#elif defined (linux)
 # include "ace/config-linux.h"

#elif defined (_MSC_VER) || defined (__BORLANDC__) || defined (__MINGW32__)
 # include "ace/config-win32.h"

#elif defined (__Lynx__)
 # include "ace/config-lynxos.h"
#endif
```

Once the above configuration is complete, build a static version of ACE, gperf, and tao_idl. To do this, first generate a minimal set of makefiles with MPC.

```
cd $ACE_ROOT/TAO
../bin/mwc.pl -type gnuace TAOACE.mwc
```

Now, build the ACE library by running GNU Make in $ACE_ROOT/ace with the ACE target:

```
cd $ACE_ROOT/ace
make ACE
```
The `gperf` and `tao_idl` executables can be built similarly by running GNU Make with no parameters in the `$ACE_ROOT/apps/gperf/src` and `$ACE_ROOT/TAO/TAO_IDL` directories.

```
cd $ACE_ROOT/apps/gperf/src
make

cd $ACE_ROOT/TAO/TAO_IDL
make
```

Now, you can reconfigure your source tree to build for your target. First, clean up libraries and object files left from the static build for your host:

```
cd $ACE_ROOT/ace
make realclean
```

Next, modify your `platform_macros.GNU` file in `$ACE_ROOT/include/makeinclude` as described in the native compilation section. An additional variable, `LYNXTARGET`, must be set when you cross-compile ACE and TAO for LynxOS. The value of this variable will be either x86 or ppc, depending upon your target platform. For example, your `platform_macros.GNU` file would appear as follows for cross-compilation for a PowerPC target:

```
VERSION = 3.1.0
LYNXTARGET = ppc
include $(ACE_ROOT)/include/makeinclude/platform_lynxos.GNU
```

If your cross-development kit is not installed in `/usr/lynx/$(VERSION)/$(LYNXTARGET)`, then you need to set `ENV_PREFIX` to the full path of the cdk either as an environment variable or in your `platform_macros.GNU` file.

```
ENV_PREFIX = /home/elliottc/lynx/$(VERSION)/$(LYNXTARGET)
```

At this point, you can follow the instructions in C.1.4, “Generate Makefiles” and finish with the remaining instructions in Appendix C.

---

**Note** You may run into difficulties building ACE and TAO with optimization turned on. The gcc 2.9-gnupro-98r2 compiler may produce an internal compiler error while compiling TAO. If this is the case, you will need to set `optimize=0` in your `platform_macros.GNU` file.
Appendix H

Testing ACE and TAO on VxWorks and LynxOS

Once you have built ACE and TAO for VxWorks or LynxOS, it is a good idea to build the tests for ACE, TAO, and orbsvcs. These tests can be useful in determining possible run-time problems that will affect your application. You can use individual tests that focus on particular features of ACE, TAO, and the orbsvcs to narrow down any particular problems you might have with your build or your environment. Not all tests will pass on both VxWorks and LynxOS.

H.1 Building the Tests

Run MPC with the following commands to generate GNU makefiles:

```
cd $ACE_ROOT/tests
$ACE_ROOT/bin/mwc.pl -type gnuace -recurse -hierarchy

cd $ACE_ROOT/TAO/tests
$ACE_ROOT/bin/mwc.pl -type gnuace -recurse -hierarchy

cd $ACE_ROOT/TAO/orbsvcs/tests
$ACE_ROOT/bin/mwc.pl -type gnuace -recurse -hierarchy
```
Run GNU Make in the following directories to build the tests:

```
$ACE_ROOT/tests
$ACE_ROOT/TAO/tests
$ACE_ROOT/TAO/orbsvcs/tests
```

This can be done very easily with the following commands:

```
cd $ACE_ROOT
make -C tests
make -C TAO/tests
make -C TAO/orbsvcs/tests
```

## H.2 Running the Tests

### H.2.1 ACE Tests

A script is provided to non-interactively run the ACE tests on UNIX, Windows, and LynxOS. However, there is currently no script to run the tests on VxWorks. On VxWorks, you will have to run the tests manually.

The automated test script runs the ACE tests that apply to the platform being tested. Output from each test is stored in a separate file in the $ACE_ROOT/tests/log directory. These files can be analyzed later for test bugs or other run-time problems.

- To run the ACE tests on LynxOS, change to the $ACE_ROOT/tests directory and run the perl script, `run_test.pl`.

- To run the ACE tests for VxWorks, you must manually load the test you want to run using `ld` from `windsh` and run it by calling the `spa` function with `ace_main` as the first argument.

The following tests in $ACE_ROOT/tests provide pretty good coverage of ACE features:

- Basic_Types_Test
- Cached_Allocator_Test
- Collection_Test
- Date_Time_Test
- Dynamic_Priority_Test
- High_Res_Timer_Test
H.2 Running the Tests

INET_Addr_Test
MT.SOCK_Test
OS_Test
Object_Manager_Test
Priority_Task_Test
Reactor_Notify_Test
Reactors_Test
SOCK_Test
Task_Test
Thread_Mutex_Test
ARGV_Test
MT_Reactor_TIMER_Test

H.2.2 TAO Tests

The TAO tests are less automated than the ACE tests. Therefore, the tests must be selectively run and the output checked by hand. Here are some of the more important tests to run, all found in $TAO_ROOT/tests:

Hello
IORManipulation
InterOp-Naming
Leader_Followers
MT_Client
MT_Server
NestedUpcall
  MT_Client_Test
  Simple
  Triangle_Test
ORB_init
OctetSeq
POA
  Current
  Default_Servant
  Etherealization
  Excessive_Object_Deactivations
  Identity
  MT_Servant_Locator
  Nested_Non_Servant_Upcalls
  Non_Servant_Upcalls
  Object_Reactivation
  Persistent_ID
  POA_Destruction
  Policies
  Single_Threaed_POA
  wait_for_completion
Param_Test
Timeout
Most TAO tests have a perl script, `run_test.pl`, that makes them a little easier to run on LynxOS. To run an individual test, change to the specific test directory and execute the `run_test.pl` script.

Testing on VxWorks can be performed in numerous ways. Since most tests consist of one or more client processes and one or more server processes, they cannot be run on VxWorks as written. We will discuss three possible ways to run these tests:

- Run the client on the target and the server or servers on the host machine.
  
  Both machines may need access to input and output files. So remember, unless you have enabled NFS in the VxWorks kernel or have a target local file system, the file system that the tasks see is that of the machine on which the target server is running.

- Run the server on the target and the client or clients on the host machine.
  
  This is similar to the above method.

- Combine the client or clients and server or servers into one module and run on the target.

  This will require writing a main function to spawn threads for each client and server.

Look at each test’s `run_test.pl` script to see the order in which the test’s programs should be run and the command line options they require. Many TAO tests depend upon the Naming Service, which can be run on the host or another machine. Many of the `orbsvcs` tests rely on multicast discovery of services; you will need to skip these tests if you do not have multicast routing enabled.

Each test sends its output to the terminal. For VxWorks, you may want to redirect the terminal to your wind shell. This can be accomplished by placing the following `tclsh` code in your `~/.wind/windsh.tcl` file:

```tcl
#Set stdin, stdout, stderr, and logging output to /vio/0 if not already in use
if { ![shParse [tstz = open("/vio/0",2,0)]] != -1 } {
    shParse { vioFd = tstz; }
    shParse { ioGlobalStdSet(0,vioFd) ;}
    shParse { ioGlobalStdSet(1,vioFd) ;}
    shParse { ioGlobalStdSet(2,vioFd) ;}
    shParse { logFdAdd(vioFd) ;}
```

Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University.
Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
Remember to close the file descriptor before you quit the wind shell. Otherwise, output may be lost since the descriptor is still open but is not attached to a terminal.

**H.2.3 ORB Services Tests**

As with the TAO tests, the orbsvcs tests are less automated than the ACE tests. Most tests have a perl script, `run_test.pl`, just as the TAO tests. The following tests, found in `$TAO_ROOT/orbsvcs/tests`, are built by default. Of course, the tests you run will depend upon which services you have chosen to build.

- Simple_Naming
- EC_Multiple
- EC_Throughput
- EC_Mcast
- EC_Custom.Marshal
- EC_Basic
- Property
- Time
- Event
  - Basic
  - Performance
- CosEvent
  - Basic
- ImplRepo
- Trading

*Note*  The ImplRepo test depends upon the ability to spawn processes, thus it does not apply to VxWorks.
Appendix I

CORBA Compliance

I.1 Introduction

This appendix contains detailed information about TAO’s compliance with various OMG CORBA specifications. The information in this appendix should not be considered as a legally binding statement. It reflects the intent of the developers of TAO as based on their interpretation of the various OMG specifications. Presently, no generally recognized acceptance tests exist for the OMG specifications listed in this appendix.

Whereas compliance with OMG specifications is a design goal of TAO, it is also a continuing pursuit. As a user of TAO, you can help in this pursuit by contacting us if you feel TAO’s implementation misinterprets any part of the cited OMG specifications, or even better, by contributing ideas and source code for how TAO’s compliance could be improved.

Note Throughout this document, “TAO” and “TAO 1.4a” refer to “OCI’s Distribution of TAO, Version 1.4a.”
CORBA Compliance

TAO is designed to be compliant with several OMG specifications. In this appendix, we address TAO’s compliance with the following OMG specifications:

- CORBA 3.0 (Core, Interoperability, Interworking, and Quality of Service)
- Minimum CORBA
- Real-Time CORBA
- CORBA C++ Language Mapping
- Naming Service
- Notification Service
- Security Service
- Data Distribution Service
- CORBA Component Model

In addition, we address TAO’s compliance with the following U.S. Defense Information Systems Agency (DISA) specification:

- Defense Information Infrastructure Common Operating Environment (DII COE)

The following sections list the exact features of each of the above specifications that TAO supports as well as describing where TAO deviates from each specification.

I.2 CORBA 3.0

TAO 1.4a is mainly compliant with the OMG CORBA 3.0.3 Core specification (OMG Document formal/04-03-12). Some areas of TAO, which have not implemented the new CORBA 3.0 functionality, remain compliant with the CORBA 2.6 specification (OMG Document formal/01-12-35). The CORBA Core specification defines four separate compliance points:

- CORBA Core
- CORBA Interoperability
- CORBA Interworking
- CORBA Quality of Service
The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one language mapping. See I.5 for information about TAO’s C++ language mapping compliance. We address TAO’s compliance with CORBA Core, CORBA Interoperability, CORBA Interworking, and CORBA Quality of Service in the separate sections below.

I.2.1 CORBA Core

TAO supports the CORBA 3.0 Core specification with the following exceptions:

- The IDL keyword `import` is not supported.
- Use of request contexts (CORBA 3.0 Chapter 3) is not supported.
- The `POAManagerFactory` interface and related features are not supported (CORBA 3.0 Section 11.3.3). The POA Manager does not support the `get_id()` operation and the POA does not support the `the_POAManagerFactory` attribute.
- IDL fixed data types (CORBA 3.0 Chapter 3) are not supported.
- Domain Managers (CORBA 3.0 Chapter 4) are not supported.
- Support for value types (CORBA 3.0 Chapter 5) and abstract interfaces (CORBA 3.0 Chapter 6) is nearly complete; exceptions are listed in 11.7.

I.2.2 CORBA Interoperability

An ORB is considered to be interoperability-compliant when it meets the following requirements:

- In the CORBA Core part of the specification, standard APIs are provided by an ORB to enable the construction of request-level inter-ORB bridges. APIs are defined by the Dynamic Invocation Interface (DII) (CORBA 3.0 Chapter 7), the Dynamic Skeleton Interface (DSI) (CORBA 3.0 Chapter 8), and by the object identity operations described in the Interface Repository chapter (CORBA 3.0 Chapter 10).
- The Internet Inter-ORB Protocol (IIOP) (CORBA 3.0 Chapter 15) defines a transfer syntax and message formats—described independently as the General Inter-ORB Protocol (GIOP)—and defines how to transfer messages via TCP/IP connections. The IIOP can be supported natively or via a half-bridge.
TAO fully supports the DII and DSI APIs. TAO also fully supports the object identity operations, such as `CORBA::Object::_is_equivalent()` and `CORBA::Object::_is_a()`. TAO implements IIOP as a pluggable protocol, and is interoperable with ORBs that support IIOP version 1.0, 1.1, or 1.2, including bi-directional GIOP/IIOP (CORBA 3.0 Chapter 15).

TAO supports the dynamic management of any values as described in the CORBA specification (CORBA 3.0 Chapter 9) with the exception of the `DynamicAny::DynAnyFactory` interface which is missing the `create_dyn_any_without_truncation` and `create_multiple_dyn_anys` operations.

CORBA 3.0 defines a new version of IIOP, version 1.3, that allows the existing IIOP 1.2 messages to be used with the component-related types. While TAO 1.4a does not support IIOP 1.3, it supports the intended component-related functionality with its IIOP 1.2 implementation.

### 1.2.3 CORBA Interworking

The purpose of the Interworking architecture is to specify support for two-way communication between CORBA objects and COM objects. The goal is for objects from one object model to be able to be viewed as if they existed in the other object model. For example, a client working in a CORBA model should be able to view a COM object as if it were a CORBA object. Likewise, a client working in a COM object model should be able to view a CORBA object as if it were a COM object.

TAO does not support the CORBA Interworking architecture specification.

CORBA Interworking also defines the Portable Interceptor facilities (CORBA 3.0 Chapter 21). TAO implements the majority of the interfaces and features described in this chapter, with the following exceptions:

- IOR interceptors are not completely implemented.
  - The `ObjectReferenceFactory` interface does not support the `equals()` and `make_profiles()` operations.
  - The `IORInterceptor_3_0` interface is not supported (but its operations are supported in the `IORInterceptor` interface).
I.2.4 CORBA Quality of Service

Quality of Service (QoS) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that the application developer need indicate only what is wanted rather than how this QoS should be achieved. Generally speaking, QoS is comprised of several QoS policies. Each QoS policy is an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to greater flexibility in CORBA applications.

TAO supports the Quality of Service portions of the CORBA 3.0 specification as follows:

- TAO fully supports the policy management framework (CORBA 3.0 Chapter 22).
- TAO implements a subset of the Messaging QoS policies, such as synchronization scope (SyncScope) for oneway requests and relative round-trip time-outs (RelativeRoundtripTimeout), and extends the Messaging QoS policies with the addition of policies for connection time-out and oneway buffering constraints (see Chapter 7).
- TAO implements the Asynchronous Method Invocation (AMI) callback model, but not the polling model (see Chapter 7).
  - TAO’s AMI implementation does not support the generic Messaging::ExceptionHolder valuetype of CORBA 3.0.
    TAO’s exception holders are generated by the IDL compiler as specified in the CORBA 2.6 specification.
- TAO does not support Time-Independent Invocation (TII).

I.3 Minimum CORBA

TAO implements the minimumCORBA specification defined in OMG Document formal/02-08-01.pdf. The minimumCORBA specification defines a profile (or subset) of CORBA that is designed for systems with limited resources. It attempts to satisfy the resource constraints of such systems while preserving portability, interoperability, and full IDL support. Support for minimumCORBA is disabled by default when TAO is built. The preprocessor
macro TAO_HAS_MINIMUM_CORBA is used to enable/disable minimumCORBA support when building TAO.

Note Enabling minimumCORBA support in TAO disables support for CORBA Messaging and Real-Time CORBA by default. CORBA Messaging can be enabled along with minimumCORBA by defining the preprocessor macro TAO_HAS_CORBA_MESSAGING with a value of 1.

The following features are disabled in TAO when minimumCORBA support is enabled:

• Dynamic Invocation Interface (DII).
• Dynamic Skeleton Interface (DSI).
• DynamicAny.
• Interface Repository.
• Implementation Repository.
• CORBA Messaging and AMI.
• Real-Time CORBA.
• Remote Policies.
• Interceptors.

• The following interfaces, exceptions, and types in module CORBA:
  Context and ContextList  
  Request and RequestSeq  
  ServerRequest  
  ConstructionPolicy  
  NamedValue, NVList, and NameValuePair  
  WrongTransaction  
  AnySeq  
  DynAny, DynSequence, DynStruct, etc.  
  FieldName  
  ORB::InconsistentTypeCode

• The following operations on CORBA::Object:
  _non_existent()  
  _get_implementation()  
  _get_interface()  
  _create_request()  
  _request()
I.4 Real-Time CORBA

- PortableServer::ForwardRequest.
- Support for pluggable protocols other than IIOP.

In addition, the following advanced POA features in the PortableServer module are disabled by default when minimumCORBA support is enabled:

- The following CORBA policies:
  - ThreadPolicy
  - ImplicitActivationPolicy
  - ServantRetentionPolicy
  - RequestProcessingPolicy
- Servant Managers (ServantActivator and ServantLocator).
- Default Servant.
- Adapter Activators.
- The following POA Manager operations and associated POA Manager states:
  - hold_requests()
  - discard_requests()
  - deactivate()
- The following UserException types in interface PortableServer::POA:
  - AdapterInactive
  - NoServant

**Note** The above advanced POA features can be enabled, even if minimumCORBA is enabled, by defining the preprocessor macro TAO_HAS_MINIMUM_POA with a value of 0 when building TAO.

I.4 Real-Time CORBA

TAO implements the Real-Time CORBA version 1.2 specification (OMG Document formal/05-01-04). Real-Time CORBA is an optional set of extensions to CORBA tailored to equip ORBs to be used as a component of a real-time system. The goals of the specification are to support developers in meeting real-time requirements by facilitating the end-to-end predictability of activities in the system and by providing support for the management of resources.
The Real-Time CORBA specification defines one mandatory compliance point, as defined in Appendix A of the Real-Time CORBA 1.2 specification. The Real-Time CORBA Dynamic Scheduling, defined in Chapter 3 of the Real-Time CORBA 1.2 specification, is a separate and optional compliance point. TAO supports Real-Time CORBA Dynamic Scheduling.

Support for Real-Time CORBA is enabled by default when TAO is built. The preprocessor macro \texttt{TAO\_HAS\_RT\_CORBA} is used to enable/disable Real-Time CORBA support when TAO is built. Real-Time CORBA support is dependent upon CORBA Messaging support.

TAO supports all of the features of the mandatory portion of the Real-Time CORBA 1.2 specification with the following exceptions:

- Priority transforms are not supported.
- Threadpool request buffering is not supported.
- Thread borrowing among threadpool lanes is not supported.

TAO extends the Real-Time CORBA 1.2 specification in the following ways:

- Addition of a Priority Mapping Manager.
- Addition of named mutexes.
- Use of a reactor-per-lane threadpool model.
- RT CORBA protocol configuration in TAO supports the selection and configuration of certain TAO pluggable protocols.
- \texttt{TCPProtocolProperties} are extended to allow the application to enable network priorities via Diffserv code points (DSCP).

See Chapter 9 for more information about using the features of the Real-Time CORBA specification with TAO.

I.5 \textbf{C++ Language Mapping}

TAO is compliant with the CORBA C++ Language Mapping version 1.1 specification (OMG Document formal/03-06-03) except for:

- Elements of the CORBA Core specification that are not supported as described in I.2.1 (i.e., abstract interfaces, fixed data types).
I.6 Naming Service

- Servant reference counting is not enabled in `PortableServer::ServantBase`. The servant base has reference counting disabled as in previous versions of the C++ mapping.

In addition, TAO supports alternative mappings for modules and exceptions, as described in the CORBA C++ language mapping specification.

I.5.1 Modules

In addition to the standard mapping for modules, TAO supports the alternative mappings for C++ dialects that do not support the `namespace` construct, as defined in 1.43.1 of OMG Document formal/03-06-03.

I.5.2 Exceptions

In addition to the standard mapping for exceptions, TAO supports the alternative mapping for C++ dialects that do not support real C++ exceptions, as defined in 1.43.2 of OMG Document formal/03-06-03. See Chapter 6 for more information about exceptions and error handling in TAO.

I.6 Naming Service

TAO implements the Naming Service version 1.2 specification (OMG Document formal/02-09-02). In addition to the `IOR`, `corbaloc`, and `corbaname` ObjectURL formats, TAO supports the following ObjectURL schemes:

- `file`
- `iiop`
- `mcast`

and optionally supports the following object URL schemes for pluggable protocols implemented by TAO (though the `corbaloc` ObjectURL scheme is preferred):

- `uiop`
- `shmiop`
- `diop`
- `uipmc`
CORBA Compliance

- sciop
- htiop

The iiop, uiop, shmiop, diop, uipmc, sciop, and htiop schemes are based on pluggable protocol implementations that are supplied with TAO. TAO supports the addition of custom pluggable transport protocols and their associated object URL schemes. See Chapter 16 and Chapter 15 for more information on pluggable protocols.

I.7 Notification Service

TAO implements the Notification Service version 1.1 specification (OMG Document formal/04-10-13) with the following exceptions:

- Only the push model is supported; the pull model is not supported.
- Mapping filters are not supported.
- Only structured and untyped events are supported; typed events are not supported.
- Filtering of untyped events is only partially supported.
- Filtering supports both the Trader Constraint Language (TCL) and the Extended Trader Constraint Language (ETCL).
- All standard Quality of Service properties are supported except for:
  - StartTime
  - StopTime
  - StartTimeSupported
  - StopTimeSupported
- Some of the standard QoS properties are not applicable at all levels defined by the specification (see 27.4.6).
- The Event Type Repository (which is an optional compliance point) is not supported.

See Chapter 27 for more information on using TAO’s Notification Service. See also $TAO_ROOT/docs/releasenotes/notify.html.
I.8 Security Service

The implementation of CORBA Security that is in the version of TAO from the DOC Group that forms the basis for OCI’s TAO 1.4a is in a state of transition. Previous versions of TAO included a partial implementation of CORBA Security and SSLIOP. The implementation in TAO 1.4a is based on newer versions of the CORBA Security specification. Unfortunately, the implementation is incomplete, so some features are not available. Further, some features that were available in previous releases may not be available as the implementation currently exists. Specifically, Policy Enforcing applications will very likely not work properly, though all Security Unaware and many Policy Changing applications will work.

See Chapter 29 for more information on using TAO’s implementation of CORBA Security.

I.9 Data Distribution Service

TAO implements the Data Distribution Service for Real-Time Systems version 1.0 specification (OMG Document formal/04-12-02). This specification defines five compliance points:

1. Minimum Profile
2. Content-Subscription Profile
3. Persistence Profile
4. Ownership Profile
5. Object Model Profile

The TAO DDS implementation is new to TAO 1.4a and is moving towards the Minimum Profile. As of this writing it is still lacking a number of features in the minimum profile. For more details on DDS and its current feature set, see Chapter 31.

I.10 CORBA Component Model

CIAO implements Version 3.0 of the OMG CORBA Component Model (CCM) (OMG Document formal/02-06-65). This specification defines Basic
and Extended conformance levels for CCM implementations. See section 1.12 of the CCM specification for more details on conformance levels. CIAO is conformant at the Basic level of conformance, with the primary exception of transaction and persistence-related CCM behavior (because TAO lacks a Transaction service). Specifically:

- CIAO does not implement the full `CCMContext` interface; it only implements `CCMContext::get_CCM_home()` (the remaining methods are transaction-related)
- CIAO does not implement the `EntityContext` interface
- CIAO does not implement the `Transactions::UserTransaction`, `SessionSynchronization`, or `EntityComponent` interfaces.
- CIAO's CIDL compiler does not support the OMG Persistent State Definition Language (PSDL).

CIAO also supports the “Deployment and Configuration of Component-based Distributed Applications” specification (OMG Document ptc/03-07-08), which supersedes the “Packaging and Deployment” chapter of the OMG CORBA Component Model (CCM) (formal/02-06-65) specification. While CIAO implements the `Packaging and Deployment` features, these interfaces are deprecated and unsupported and should not be used.

I.11 DII COE

The Defense Information Infrastructure Common Operating Environment (DII COE) is a set of specifications, defined by the U.S. Defense Information Systems Agency (DISA), for developing and packaging software used by defense and intelligence agencies. In addition to these specification, DII COE also defines processes for requirements and systems engineering, product selection/submission, testing, and configuration management. Finally, DII COE provides a repository of commonly-useful software and data components developed by both the commercial sector and the government that have received DII COE certification. The armed services and other DII COE participants can use this repository to guide their buying decisions.

The DII COE computing model comprises a three-tiered architecture:

- The `kernel` layer consists of certified hardware and operating system platforms.
The **infrastructure services** layer consists of administrative, communications, distributed computing, security, and data access/object management services.

The **common support applications** layer includes APIs for such common application functions as alerts, message passing, correlation, office automation, multimedia and collaboration, on-line help, and data presentation. Individual applications operate on top of the common support applications layer.

DII COE includes a certification authority and various change control boards. Applicants perform assessments of their own products against DII COE certification criteria and submit them to the DII COE certification authority for inclusion in the DII COE application repository.

The Center for Distributed Object Computing (the “DOC group”) at Washington University in St. Louis, Missouri, performed a DII COE assessment for ACE version 5.2 and TAO version 1.2. Five components of ACE and TAO were assessed on each of three DII COE qualified platforms—Solaris 8 Update 1, Windows NT SP6, and LynxOS 3.1.0a—for a total of fifteen assessments. The following components were assessed:

- PACE Library Extended Toolkit
- ACE Library Extended Toolkit
- TAO ORB Library Extended Toolkit
- TAO Naming Service
- TAO Real-Time Event Service

Each of these assessments passed all eight levels of DII COE compliance criteria.

No additional DII COE assessments have been performed since TAO 1.2. Note that the PACE library is no longer part of ACE and TAO.

Additional information on DII COE can be found at <http://www.disa.mil/coe/>. Additional information on the details of ACE and TAO compliance can be provided by OCI upon request.
Appendix J

TAO and ACE Contributors

TAO and ACE, though primarily developed and maintained by members of the “DOC group”<http://www.dre.vanderbilt.edu/>, is a truly open source project. Well over a thousand people have contributed to it in one way or another, such as adding new features, reporting bugs, contributing patches, porting to new platforms, or adding documentation.

All of the (known) contributors are listed here, in alphabetical order. (Some were modest and used nom de plumes or just their first names. Let us know if you would like full credit.) Our apologies for misspelling or otherwise misrepresenting anyone’s name. Current and former members of the “DOC groups” at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, the Center for Distributed Object Computing at Washington University, the Laboratory for Distributed Object Computing at the University of California, Irvine, and members of the OCI staff are indicated in boldface.

Our thanks to all of you for making TAO and ACE a success!

Sami Aario
Chris Able
Manojkumar Acharya
TAO and ACE Contributors

Joachim Achtzehnter
Donald Acton
Alexei I. Adamovich
Jeff Adams
Mick Adams
Matthew Adams
Frank Adcock
Marc M Adkins
Sandeept Adwankar
Jamshid Afshar
Giulio Agostini
Sasha Agranov
Newton Aird
Walt Akers
Knut-Havard Aksnes
Bruce Alderson
Syed Wasim Ali
David Allen
Jeremy Altavilla
Marc Alter
Eugene Alterman
Michael Altmann
Andy Alvarez
Sal Amander
Virginie Amar
Karen Amestoy
Tomer Amiaz
Michele Amoretti
Jingbin An

**Everett Anderson (DOC)**

Craig Anderson
Mike Anderson
Lori Anderson
Doug Anderson
Per Andersson
Tommy Andreasen
Frederic Andres
Rob Andzik
Bernd Annamaier
Brian Appel
Gheorghe Aprotosoaie
Tom Arbuckle
Yaniv Ben Ari
Valery Arkhangorodsky
David Arndt
Hugh Arnold

Diana Arroyo
David Artus
**Alexander Babu**
**Arulanthu (DOC)**
Zvika Ashani
John Ashmun
Jonathan Astle
Vijay Aswadhati
Andrew Athan
Raja Ati
**Shawn Atkins (DOC)**
Alia Atlas
John Aughey
Pradeep Avasthi
Edan Ayal

**Vital Aza (DOC)**
Hamed Azizadah
Krishnakumar B.
Heiko Bachmann
Torbjorn Backstrom
Bill Backstrom
Amir Bahmanyari
Wenli Bai
Lee Baker
Greg Baker

**Krishnakumar**

**Balasubramanian (DOC)**

Bo Balder
Francesco Baldi
Craig Ball
George Ball
Koushik Banerjee
Alex Bangs
Sush Bankapura
Klaus Banzer
Ami Bar
Ron Barack
Ron Barack
Shannon Barber
Veron Solrzano Barboza
Christian Barheine
Todd Barkalow
Mark C. Barnes
Shameek Basu

Vyacheslav A. Batenin
Viatcheslav Batenev
Mateu Batle
Alan I Batongbacal
Bonifides Bautista
James Beale
Roger Beathard
Emmanuel Thevenot
Beaufort
Roger Beck
Detlef Becker
N Becker
Didier Becu
Paul von Behren
Peter Bekiesch
Daniel Bell
Andy Bellalafaire
Dmitri Belogaj
Alexander Belopolsky
Ram Ben-Yakir
Raz Ben-Yehuda
Manuel Benche
Patrick Bennett
Patrice Bensousan
Emiliano Berenbaum
Tobin Bergen-Hill
Ronald Berger
Mike Bernat
Andrea Bernicchia
Francois Bernier
Udo Berninger
Geir Berset
Eric Beser
EvgenyBeskovnny
Stephan Bettermann
Shalabh Bhatnagar
Probal Bhattacharjya
Shashi Bhushan
Jerry Bickle
Harold Bien
Jonathan Biggar
Trueman Bill
Max F. Bilyk
Duane Binder
Girish Birajdar
Pierre Bisaillon
<table>
<thead>
<tr>
<th>TAO and ACE Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guicheney Christophe</td>
</tr>
<tr>
<td>Rich Christy</td>
</tr>
<tr>
<td>Bill Church</td>
</tr>
<tr>
<td>Rob Clairmont</td>
</tr>
<tr>
<td>Dorr H. Clark</td>
</tr>
<tr>
<td>Graeme Clark</td>
</tr>
<tr>
<td>Bret Clark</td>
</tr>
<tr>
<td>Scott Clarke</td>
</tr>
<tr>
<td><strong>Chris Cleeland</strong></td>
</tr>
<tr>
<td>(DOC/OCI)</td>
</tr>
<tr>
<td>Guicheney Christophe</td>
</tr>
<tr>
<td>Rich Christy</td>
</tr>
<tr>
<td>Bill Church</td>
</tr>
<tr>
<td>Rob Clairmont</td>
</tr>
<tr>
<td>Dorr H. Clark</td>
</tr>
<tr>
<td>Graeme Clark</td>
</tr>
<tr>
<td>Bret Clark</td>
</tr>
<tr>
<td>Scott Clarke</td>
</tr>
<tr>
<td><strong>Angelo Corsaro</strong></td>
</tr>
<tr>
<td>(DOC)</td>
</tr>
<tr>
<td>Guicheney Christophe</td>
</tr>
<tr>
<td>Rich Christy</td>
</tr>
<tr>
<td>Bill Church</td>
</tr>
<tr>
<td>Rob Clairmont</td>
</tr>
<tr>
<td>Dorr H. Clark</td>
</tr>
<tr>
<td>Graeme Clark</td>
</tr>
<tr>
<td>Bret Clark</td>
</tr>
<tr>
<td>Scott Clarke</td>
</tr>
<tr>
<td><strong>Eric Crampton</strong></td>
</tr>
<tr>
<td>(DOC/OCI)</td>
</tr>
<tr>
<td>Guicheney Christophe</td>
</tr>
<tr>
<td>Rich Christy</td>
</tr>
<tr>
<td>Bill Church</td>
</tr>
<tr>
<td>Rob Clairmont</td>
</tr>
<tr>
<td>Dorr H. Clark</td>
</tr>
<tr>
<td>Graeme Clark</td>
</tr>
<tr>
<td>Bret Clark</td>
</tr>
<tr>
<td>Scott Clarke</td>
</tr>
<tr>
<td><strong>Enrico Detoma</strong></td>
</tr>
<tr>
<td>Thomas Devanneaux</td>
</tr>
<tr>
<td>Gonzalo Diethelm</td>
</tr>
<tr>
<td>David Digby</td>
</tr>
<tr>
<td><strong>Jianu Ding (DOC)</strong></td>
</tr>
<tr>
<td>(OCI)</td>
</tr>
<tr>
<td>Guicheney Christophe</td>
</tr>
<tr>
<td>Rich Christy</td>
</tr>
<tr>
<td>Bill Church</td>
</tr>
<tr>
<td>Rob Clairmont</td>
</tr>
<tr>
<td>Dorr H. Clark</td>
</tr>
<tr>
<td>Graeme Clark</td>
</tr>
</tbody>
</table>

Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.
<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roger Egbers</td>
<td>Mike Flinn</td>
<td>John Geiss</td>
</tr>
<tr>
<td>Christian Egeler</td>
<td>Brad Flood</td>
<td>Martin Geliot</td>
</tr>
<tr>
<td>Rob Eger</td>
<td>Sergio Flores-Gaitan (DOC)</td>
<td>Patty Genualdi</td>
</tr>
<tr>
<td>Chris Eich</td>
<td>Karim Fodil-Lemelin</td>
<td>Peter Georgakakis</td>
</tr>
<tr>
<td>Eric Eide</td>
<td>Andre Folkers</td>
<td>William L. Gerecke</td>
</tr>
<tr>
<td>Preston Elder</td>
<td>Andre Folkers</td>
<td>Soeren Gerlach</td>
</tr>
<tr>
<td>Kirk Ellett</td>
<td>John Foresteire</td>
<td>Artashes Ghazaryan</td>
</tr>
<tr>
<td>Chad Elliott (OCI)</td>
<td>Harry Forry</td>
<td>Fabrizio Giannotti</td>
</tr>
<tr>
<td>Vicentini Emanuele</td>
<td>Michael Fortinsky</td>
<td>Roland Gigler</td>
</tr>
<tr>
<td>Matt Emerson</td>
<td>Andrew Foster</td>
<td>Giga Giguashvili</td>
</tr>
<tr>
<td>Ben Eng</td>
<td>Paul Francis</td>
<td>Dan Gilboa</td>
</tr>
<tr>
<td>Marc Eng</td>
<td>Ruediger Franke</td>
<td>Chris Gill (DOC)</td>
</tr>
<tr>
<td>Richard Eperjesi</td>
<td>Jonathan Franklin</td>
<td>Matthew Gillen</td>
</tr>
<tr>
<td>Caleb Epstein</td>
<td>Jeff Franks</td>
<td>Kim Gillies</td>
</tr>
<tr>
<td>Wayne Erchak</td>
<td>Steven Frare</td>
<td>Michael Gillmann</td>
</tr>
<tr>
<td>Jack Erickson</td>
<td>Charles Frasch</td>
<td>Brian Gilmer</td>
</tr>
<tr>
<td>Stefan Ericsson</td>
<td>Stephan Frenzel</td>
<td>Andrew Gilpin (DOC)</td>
</tr>
<tr>
<td>Mattias Eriksson</td>
<td>Zach Frey</td>
<td>Brian Gilstrap (OCI)</td>
</tr>
<tr>
<td>Letha Etzkorn</td>
<td>Eric Frias</td>
<td>Mike Gingell</td>
</tr>
<tr>
<td>J dot Scott Evans</td>
<td>Hamish Friedlander</td>
<td>Thomas Girard</td>
</tr>
<tr>
<td>Mark Evans</td>
<td>Mark Frutig</td>
<td>Virgilijus Globis</td>
</tr>
<tr>
<td>Christian Ewald</td>
<td>Benjamin Fry</td>
<td>John Glynn</td>
</tr>
<tr>
<td>Michael F&quot;olsl</td>
<td>Joyce Fu</td>
<td>Sergey Gnilitsky</td>
</tr>
<tr>
<td>Adam Fanello</td>
<td>Bill Fulton</td>
<td>Aniruddha Gokhale (DOC)</td>
</tr>
<tr>
<td>Ayman Farahat</td>
<td>Wai Keung Fung</td>
<td>Mike Goldman</td>
</tr>
<tr>
<td>Robin Farine</td>
<td>Bhaskara Rao G</td>
<td>Dmitry Goldshtain</td>
</tr>
<tr>
<td>David Faure</td>
<td>Scott Gaa</td>
<td>Mickael P. Golovin</td>
</tr>
<tr>
<td>Pierre Fayolle</td>
<td>Rob Gabbot</td>
<td>Israel Illescas Gomez</td>
</tr>
<tr>
<td>Gregory D. Fee (DOC)</td>
<td>Ildar Gabdulline</td>
<td>Priyanka Gontla (DOC/OCI)</td>
</tr>
<tr>
<td>Pedro Alves Ferreira</td>
<td>Alexey Gadzhiev</td>
<td>Richard Goold</td>
</tr>
<tr>
<td>Cristian Ferretti</td>
<td>Christophe Galerne</td>
<td>Pradeep Gore (DOC)</td>
</tr>
<tr>
<td>Petr Ferschmann</td>
<td>Greg Gallant</td>
<td>Peter Gorgia</td>
</tr>
<tr>
<td>Trevor Fields (OCI)</td>
<td>Slava Galperin</td>
<td>Vasili Goutas</td>
</tr>
<tr>
<td>Evghenii Filippov</td>
<td>Jaymes Galvin</td>
<td>Sandeep Goyal</td>
</tr>
<tr>
<td>Andrew Finnell</td>
<td>Scott Gammil</td>
<td>Michael Graf</td>
</tr>
<tr>
<td>Harald Finster</td>
<td>Scott Gammill</td>
<td>Jeff Graham</td>
</tr>
<tr>
<td>Peter Fischer</td>
<td>Steven Gardner</td>
<td>Jeffrey Graham</td>
</tr>
<tr>
<td>Ronald Fischer</td>
<td>Sudhanshu Garg (DOC)</td>
<td>Charlie Grames</td>
</tr>
<tr>
<td>Roland Fischer</td>
<td>Daniel Garrido</td>
<td>Gerard Grant</td>
</tr>
<tr>
<td>Paul K. Fisher</td>
<td>James Garrison</td>
<td>Jeff Gray</td>
</tr>
<tr>
<td>Robert Flanders</td>
<td>Michael Garvin</td>
<td>Ishay Green</td>
</tr>
<tr>
<td>David Fleeman</td>
<td>John Gathright</td>
<td>Dan Green</td>
</tr>
<tr>
<td>Frank O. Flemisch</td>
<td>Alexandr Gavrilov</td>
<td>Marvin Greenberg</td>
</tr>
<tr>
<td>John Fletcher</td>
<td>Chuck Gehr</td>
<td>Jesse Greenwald</td>
</tr>
<tr>
<td>Dani Flexer</td>
<td>Andreas Geisler</td>
<td></td>
</tr>
<tr>
<td>Ben Flight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contributors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeff Greif</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Darren Griffith</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carl Grinstead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pierre Grondin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gilbert Grosdidier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peter Gross</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matthew Grosso</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thomas Groth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael Grove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trey Grubbs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Todd Gruhn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joe Guan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stephan Gudmundson</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stanford S. Guillory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ernesto Guisado</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ru Gunatilleke (OCI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scott Gunn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harry Gunnarsson</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alok Gupta</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J. Abelardo Gutierrez</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Johannes Gutleber</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andy Guy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andrew Guy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Martin Habets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dana Hackman</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chris Hafey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jody Hagins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>James Haier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattie Hair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dan Halbert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dave Hale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bill Hall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David Hall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greg Hall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scott Halstead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jake Hamby</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jam Hamidi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Randy Hammon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chris Hammond</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael Hampel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thomas Hampson</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Matthew P. Hampton (DOC)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zheng Han</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paul Han</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robert Handl</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shawn Hannan (DOC)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Daniel Hannum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mogens Hansen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David Hanvey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mitsuhiko Hara</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andrew Harbick</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Richard Hardgrave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Espen Harlinn</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Byron Harris (OCI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Scott Harris (OCI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matthew Harris</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tim Harrison (DOC)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greg Harrison</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael Hartman</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andrew G. Harvey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Richard G. Hash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ron Hashimshony</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amancio Hasty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David Hauck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sam Hauer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Philippe Haussy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tim Hawes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dale Hawkins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeff R. Hayes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joe Hayes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heping He</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robert Head</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lyn Headley</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ron Heald</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chris Healey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>John P. Hearn</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kevin Heifner (OCI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Randy Heiland</td>
<td></td>
<td></td>
</tr>
<tr>
<td>John E Hein</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Benedikt Eric Heinen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peter Heitman</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>John Heitmann (DOC)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eric Held</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeff Hellzen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bayu Hendrajaya</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michi Henning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stephen Henry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mike Hepburn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peter Hercek</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hanan Herzog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Steve Hespelt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rick Hess</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tad Hetke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kelly F. Hickel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>John Hickin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Steffen Hieber</td>
<td></td>
<td></td>
</tr>
<tr>
<td>John Hiltenbrand</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Don Hinton (DOC)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wada Hiroshi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long Hoang</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andrew Hobson</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erich Hochmuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lorin Hochstein</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frank J. Hodum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximilian Hoferer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Todd Hoff</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Joe Hoffert (DOC)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>William A. Hoffman</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael Hoffman</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ekkhard Hoffmann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mark Hoffmann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mario Hofmann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Klaus Hofmann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bernd Hofner</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barry Hoggard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael Hollins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greg Holtmeyer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raymond Hoofman</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nigel Hooke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>David Hooker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seung-Lee Hoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bill Hopkins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeff Hopper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eric Hopper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toshio Hori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>William Horn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alex Hornby</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael Hornok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hans Horsmann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ted Horst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derek Horton</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brad Hoskins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stephen Howard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tom Howard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ben Howard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adam Howell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dmitri Hrapof</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>James Hu (DOC)</td>
<td>Sangwoo Jin</td>
<td>Magnus Karlsson</td>
</tr>
<tr>
<td>Huang-Ming Huang (DOC)</td>
<td>Guan Joe</td>
<td>David Karr</td>
</tr>
<tr>
<td>Thomas Huang</td>
<td>Erik Johannes</td>
<td>Dmitri Katchalov</td>
</tr>
<tr>
<td>Richard Huber</td>
<td>Knut Johannessen</td>
<td>Arun Katkere</td>
</tr>
<tr>
<td>John Hutchinson</td>
<td>Ciju John (OCI)</td>
<td>Jason Katz</td>
</tr>
<tr>
<td>Andreas Huggel</td>
<td>Kristopher Johnson</td>
<td>Martin Kaul</td>
</tr>
<tr>
<td>Chris Hughes</td>
<td>James CE Johnson</td>
<td>Steve Kay</td>
</tr>
<tr>
<td>Dominic Hughes</td>
<td>Martin Johnson</td>
<td>Richard Keizer</td>
</tr>
<tr>
<td>Conrad Hughes</td>
<td>Richard L. Johnson</td>
<td>Alfred Keller</td>
</tr>
<tr>
<td>Frank A. Hunleth (DOC)</td>
<td>Craig Johnston</td>
<td>Jeff Kelley</td>
</tr>
<tr>
<td>Carol Hunsicker</td>
<td>Bob Jolliffe</td>
<td>Oliver Kellogg</td>
</tr>
<tr>
<td>Sohail Husain</td>
<td>Jeff Jones</td>
<td>Allen Kelly</td>
</tr>
<tr>
<td>Ole Husgaard</td>
<td>Chip Jones (OCI)</td>
<td>Keo Kelly</td>
</tr>
<tr>
<td>Steve Huston</td>
<td>Stuart Jones</td>
<td>Matthias Kerkhoff</td>
</tr>
<tr>
<td>Nicolas Huynh</td>
<td>Jacob Jones</td>
<td>Mark Kettner</td>
</tr>
<tr>
<td>Mark Hyett</td>
<td>Bruce Jones</td>
<td>Whitney Kew</td>
</tr>
<tr>
<td>Robert S. Iakobashvili</td>
<td>Erik Jones</td>
<td>Max Khoon</td>
</tr>
<tr>
<td>Hessel Idzenga</td>
<td>Brian Jones</td>
<td>Dmitry Khrapov</td>
</tr>
<tr>
<td>Steve Ige</td>
<td>Mark De Jong</td>
<td>Chatchai Khumboa</td>
</tr>
<tr>
<td>Soren Ilsoe</td>
<td>Thomas Jordan</td>
<td>Stefaan Kiebooms</td>
</tr>
<tr>
<td>Lars Immisch</td>
<td>Rob Jordan</td>
<td>Thilo Kielmann</td>
</tr>
<tr>
<td>Hakon Innerdal</td>
<td>Sudish Joseph</td>
<td>Timothy Kilbourn</td>
</tr>
<tr>
<td>Sorin Iordachescu</td>
<td>Rod Joseph</td>
<td>Jovan Kilibarda</td>
</tr>
<tr>
<td>Sorin Iordachescu</td>
<td>Sandeep Joshi</td>
<td>Chumsu Kim</td>
</tr>
<tr>
<td>Tim Iskander</td>
<td>Yateen Joshi</td>
<td>Dongwook Kim</td>
</tr>
<tr>
<td>Panagiotis Issaris</td>
<td>Jerome Julius</td>
<td>Jaepil Kim</td>
</tr>
<tr>
<td>Erik Ivanenko</td>
<td>Henric Junghiem</td>
<td>David Kinder</td>
</tr>
<tr>
<td>Zoran Ivanovic</td>
<td>Christophe Juniet</td>
<td>Andy King</td>
</tr>
<tr>
<td>Allan S Iverson</td>
<td>Gildo Medeiros Junior</td>
<td>Guy Bolton King</td>
</tr>
<tr>
<td>Uwe Jaeger</td>
<td>Christina Junru</td>
<td>Michael Kircher</td>
</tr>
<tr>
<td>Ulf Jaehrig</td>
<td>Vishal Kachroo (DOC)</td>
<td>Victor Kirk</td>
</tr>
<tr>
<td>Manish Jain</td>
<td>Henrik Kai</td>
<td>Tibor Kiss</td>
</tr>
<tr>
<td>Dipti Jain</td>
<td>Ashok Kumar Kalanithi</td>
<td>Jay Kistler</td>
</tr>
<tr>
<td>Prashant Jain (DOC)</td>
<td>Jan Kalin</td>
<td>Marius Kjeldahl</td>
</tr>
<tr>
<td>Leif Jakobsmeier</td>
<td>Hakan Kallberg</td>
<td>Andre Kleibeuker</td>
</tr>
<tr>
<td>David Janello</td>
<td>Natarajan Kalpathy</td>
<td>Philippe Klein</td>
</tr>
<tr>
<td>Dimitrije Jankovic</td>
<td>Boris Kaminer</td>
<td>Ron Klein</td>
</tr>
<tr>
<td>Greg Jansen</td>
<td>Leen Van Kampen</td>
<td>Stefan Kluhspies</td>
</tr>
<tr>
<td>Tad Jarosinski</td>
<td>Mike Kamrad</td>
<td>Ralf Kluthe</td>
</tr>
<tr>
<td>Alexander Jasper</td>
<td>Johann Kandlbauer</td>
<td>Christian Klutz</td>
</tr>
<tr>
<td>Joseph Jefferson</td>
<td>Ken Kane</td>
<td>Dave Knox</td>
</tr>
<tr>
<td>Iliyan Jeliazkov (OCI)</td>
<td>Bharathi Kangatharan</td>
<td>Dieter Kneuppel</td>
</tr>
<tr>
<td>Jorn Jensen</td>
<td>Suresh Kannan</td>
<td>Andreas Koehler</td>
</tr>
<tr>
<td>Bogdan Jeram</td>
<td>James Kanyak</td>
<td>Erik Koerber</td>
</tr>
<tr>
<td>Jerry Jiang</td>
<td>Srikrumar Kareti</td>
<td>Alexander Kogan</td>
</tr>
<tr>
<td></td>
<td>Alexandre Karev</td>
<td>Ran Kohavi</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Christopher Kohlhoff</td>
<td>Yury Kuznesov</td>
<td>Tom Leith</td>
</tr>
<tr>
<td>Denny Kolb</td>
<td>Slawomir Kuzniar</td>
<td>Gerhard Lenzer</td>
</tr>
<tr>
<td>Serge Kolgan</td>
<td>Leonid Kvetnyi</td>
<td>Antonio Leonforte</td>
</tr>
<tr>
<td><strong>Boris Kolpackov (DOC)</strong></td>
<td>Yuk Ming Kwok</td>
<td>Ivan Leong</td>
</tr>
<tr>
<td>Miroslav Koncar</td>
<td>Fred LaBar</td>
<td>Kim Lester</td>
</tr>
<tr>
<td>Vladimir Kondratiev</td>
<td>James Lacey</td>
<td>Mike Letchworth</td>
</tr>
<tr>
<td>Ken Konckeck</td>
<td>Larry Lachman</td>
<td>Sunny Leung</td>
</tr>
<tr>
<td>Ansgar Konermann</td>
<td>Florian Lackerbauer</td>
<td>Tal Lev-Ami</td>
</tr>
<tr>
<td>Madhu Konety</td>
<td>Terry Lacy</td>
<td>Moran Levi</td>
</tr>
<tr>
<td>Vincent Korkos</td>
<td>Bob Laferriere</td>
<td>Dan Levi</td>
</tr>
<tr>
<td>Christian Korn</td>
<td>Mark Laffoon</td>
<td></td>
</tr>
<tr>
<td>Andre Kostur</td>
<td>George Lafortune</td>
<td></td>
</tr>
<tr>
<td>Devesh Kothari</td>
<td>Chris Lahey</td>
<td></td>
</tr>
<tr>
<td>Leo Kov</td>
<td>Michael Laing</td>
<td></td>
</tr>
<tr>
<td>Jenny Kowald</td>
<td>Tom Lake</td>
<td></td>
</tr>
<tr>
<td>Norbert Krain</td>
<td>V dot Lakshmananan</td>
<td></td>
</tr>
<tr>
<td>Michael Kramer</td>
<td>Sebastien Lalonde</td>
<td></td>
</tr>
<tr>
<td>Marco Kranawetter</td>
<td>Jon Lambert</td>
<td></td>
</tr>
<tr>
<td>Nathan Krasney</td>
<td>Yaolong Lan</td>
<td></td>
</tr>
<tr>
<td>Oleg Kraynov</td>
<td>Sean Landis</td>
<td></td>
</tr>
<tr>
<td>Victor Krebss</td>
<td>Theo Landman</td>
<td></td>
</tr>
<tr>
<td>Holger P. Krekel</td>
<td>Uwe Landrock</td>
<td></td>
</tr>
<tr>
<td><strong>Arvind S. Krishna (DOC)</strong></td>
<td><strong>Andras Lang (DOC)</strong></td>
<td></td>
</tr>
<tr>
<td>Shankar Krishnamoorthy</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Yamuna Krishnamurthy (DOC)</strong></td>
<td>Daniel Lang</td>
<td></td>
</tr>
<tr>
<td>Oleg Krivosheev</td>
<td>Wilbur Lang</td>
<td></td>
</tr>
<tr>
<td>Ted Krovetz</td>
<td>Frederic Langlet</td>
<td></td>
</tr>
<tr>
<td>Olaf Kruger</td>
<td>Terry Lao</td>
<td></td>
</tr>
<tr>
<td>Martin Krumpolec</td>
<td>Joseph E. LaPrade</td>
<td></td>
</tr>
<tr>
<td>Andres Kruse</td>
<td>Patrick J Lardieri</td>
<td></td>
</tr>
<tr>
<td>Sergei Kuchin</td>
<td>Roger Larsson</td>
<td></td>
</tr>
<tr>
<td>Ugendreshwar Kudupudi</td>
<td>Elliot Lau</td>
<td></td>
</tr>
<tr>
<td>Torsten Kuepper</td>
<td>Olivier Lau</td>
<td></td>
</tr>
<tr>
<td>Frank Kuhlman</td>
<td>Dustin Laurence</td>
<td></td>
</tr>
<tr>
<td><strong>Fred Kuhns (DOC)</strong></td>
<td>Neil Lavelle</td>
<td></td>
</tr>
<tr>
<td>Nick Kukuczka</td>
<td>Greg Lavender</td>
<td></td>
</tr>
<tr>
<td>Abhay Kulkarni</td>
<td>Bill Lear</td>
<td></td>
</tr>
<tr>
<td>Peter Kullmann</td>
<td>William S. Lear</td>
<td></td>
</tr>
<tr>
<td>Stephan Kulow</td>
<td>Vianney Lecroart</td>
<td></td>
</tr>
<tr>
<td>Vinod Kumar</td>
<td>Haifeng Lee</td>
<td></td>
</tr>
<tr>
<td>Sunil Kumar</td>
<td>Yiu L. Lee</td>
<td></td>
</tr>
<tr>
<td>Surender Kumar</td>
<td>Stan Leeson</td>
<td></td>
</tr>
<tr>
<td>Mitch Kuninsky</td>
<td>Marc Lehmann</td>
<td></td>
</tr>
<tr>
<td>Amir Kunst</td>
<td>Ville Lehtiniemi</td>
<td></td>
</tr>
<tr>
<td>Kirill Kuolechov</td>
<td>Philipp Leibfried</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chris Leishman</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Philip Leishman</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**OCI and ACE Contributors**

*Licensed for use by the School of Electrical Engineering and Computer Science, Washington State University.
Not licensed for redistribution. © 2008 Object Computing, Inc. All Rights Reserved.*
<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marcel Loose</td>
<td>Ricky Marek</td>
<td>Edwin McKay</td>
</tr>
<tr>
<td>Luis Lopes</td>
<td>Konstantinos Margaritis</td>
<td>David McKen</td>
</tr>
<tr>
<td>Theckla Louchios</td>
<td>Petru Marginean</td>
<td>Michael McKnight</td>
</tr>
<tr>
<td>Nigel Lowe</td>
<td>Erik Margraf</td>
<td>Dave McNeely</td>
</tr>
<tr>
<td>Goran Lowkraftz</td>
<td>Frederic Maria</td>
<td>Pat McNerthney</td>
</tr>
<tr>
<td>Joe Loyall</td>
<td>Mark Maris</td>
<td>Jeff McNiel</td>
</tr>
<tr>
<td><strong>Tao Lu (DOC)</strong></td>
<td>Uma Markandu</td>
<td>Simon McQueen</td>
</tr>
<tr>
<td>John Lu</td>
<td>Weston Markham</td>
<td>Malcolm McRoberts</td>
</tr>
<tr>
<td>Eyal Lubetzky</td>
<td>Hajdukiewicz Markus</td>
<td>David McWeeny</td>
</tr>
<tr>
<td>Rainer Lucas</td>
<td>Rick Marlborough</td>
<td>Bob McWhirter</td>
</tr>
<tr>
<td>Casey Lucas</td>
<td>Kees van Marle</td>
<td>Eric R. Medley</td>
</tr>
<tr>
<td>Mark Lucovsky</td>
<td>Andrew Marlow</td>
<td>Stanislav Meduna</td>
</tr>
<tr>
<td>Jonathan Luellen</td>
<td>Patrice Marques</td>
<td>James Megquier</td>
</tr>
<tr>
<td>Alex Luk</td>
<td>Paul Marquis</td>
<td>Thomas Mehrkam</td>
</tr>
<tr>
<td>Gruetzmacher Lukas</td>
<td>Tom Mars</td>
<td>Charles Meidinger</td>
</tr>
<tr>
<td>Johan Lundin</td>
<td>Rohan Mars</td>
<td>Charles Meier</td>
</tr>
<tr>
<td>Mikael Lundqvist</td>
<td>Dennis C. De Mars</td>
<td>Samuel Melamed</td>
</tr>
<tr>
<td>Samuel Qi Luo</td>
<td>Todd Marshall</td>
<td>Jim Melton</td>
</tr>
<tr>
<td>Steve Luoma</td>
<td>Kevin Marshall</td>
<td>Iain Melville</td>
</tr>
<tr>
<td><strong>Robert Martin (OCI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kevin Martindale</td>
<td>Brian Mendel</td>
</tr>
<tr>
<td></td>
<td>Umberto Mascia</td>
<td>Praphul Menon</td>
</tr>
<tr>
<td></td>
<td>John Masiyowski</td>
<td>Adrian Mercieca</td>
</tr>
<tr>
<td></td>
<td>Donna Maskell</td>
<td>Peter van Merkerk</td>
</tr>
<tr>
<td></td>
<td>Peter J. Mason</td>
<td>Berni Merkle</td>
</tr>
<tr>
<td></td>
<td>Paxton Mason</td>
<td>Phillippe Merle</td>
</tr>
<tr>
<td></td>
<td>Brian Mason</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Michael R. MacFaden</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Simon Massey</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Jerry D. De Master</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Marcelo Matus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hani Mawlawi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gary Maxey</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Michael Maxie</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dave Mayerhoefer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>James Maynard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mike Mazurek</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Greg McCain</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pete McCann</td>
<td></td>
</tr>
<tr>
<td></td>
<td>David McCann</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sean McCauliff</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Anthony McConnell</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Braden McDaniel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Jeff McDaniel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Andrew McGowan</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Souhad Mcheik</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bruce McIntosh</td>
<td></td>
</tr>
<tr>
<td><strong>Phil Mesnier (OCI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Andrew Metcalfe</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Roland Meub</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dave Meyer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Satheesh Kumar MG</td>
<td></td>
</tr>
<tr>
<td><strong>Justin Michel (OCI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Christopher W. Midgley</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Terry Mihm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branko Mijic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Onopin V. Mikhail</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Warren Miller</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Philip Miller</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adam Miller</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Jason Milley</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Christian Millour</td>
<td></td>
</tr>
<tr>
<td></td>
<td>John M. Mills</td>
<td></td>
</tr>
<tr>
<td></td>
<td>John Mills</td>
<td></td>
</tr>
<tr>
<td></td>
<td>John Mink</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Alex Mintz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fernando D. Mato Mira</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adrian Miranda</td>
<td></td>
</tr>
<tr>
<td><strong>Pavan Mandalkar (DOC)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Daniel Manfis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>James Mansion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andy Marchewka</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Przemyslaw Marciniak</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bruno Marconi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Daniel Miranda
David Miron
Jeff Mirwaisi
Calum Mitchell
Eric Mitchell
Fukasawa Mitsuo
Logan Modahala
Leo Modica
Dirk Moermans
Sam Mok
Daniel Montalibet
Domingos Monteiro
Dom Monteiro
Arturo Montes
Xavier Montet
Todd L. Montgomery
Tom Moog
Stephen Moon
Masaoud T. Moonim
Dave Moore
Russell Mora
Ramiro Morales
Mike Moran (DOC)
Ciaran Moran
John Morey
Jeff Morgan
James Morris
Rodney Morris
Paul Morrison
Frederic Motte
Bertrand Motuelle
Paul Motuzenko
Pavel Motuzenko
Vince Mounts
Bin Mu
Ron Muck
Christian Mueffling
Peter Mueller
Ted Mules
John Mulhern
Edward R. Mulholland
Todd Mullanix
Schone Mullerin
Greg Mulyar
Sumedh Mungee (DOC)
Andrew Munro
Ivan Murphy
Matt Murphy
Anthony Mutiso
Stuart Myles
Jesper S. Møller
Patrick N
Suresh N
Victor N.
Peter Liqun Na
Ravi Nagabhyru
Ramesh Nagabushnam
William Nagel
Tushar Nair
Raghu Nambaiith
Priya Narasimhan
Raghu Narayan
Vidya Narayanan
Raj Narayanaswamy
Heiko Nardmann
Avi Nash
Balachandran Natarajan
(DOC)
Thomas Natterer
Vladimir Naylor
Mehrdad Nazari
Andrey Nechyporenko
Brad Needham
Kumar Neelakantan
Sandeep Neema (DOC)
Brian Nelson
Sergey Nemanov
Burkhard Neppert
Bill Nesbitt
Eyal Neuman
Matthew Newhook
Rich Newman
Vincent Newsom
Michael Newton
Eric C. Newton
Tam Nguyen
Giang Hoang Nguyen
Keith Nicewarner
Keith Nichol
Vincent Nicolas
Jan Nielsen
Carsten T. Nielsen
Erik Toubro Nielsen
Frederick Niemi
Daniel Nieten
Mats Nilsson
Mattias Nilsson
Aurelio Nocerino
Ted Nolan
Dennis Noll
Henrik Nordberg
Mike Nordell
Jonas Nordin
Peter Nordlund
Peder Norgaard
Miljenko Norsic
Neal Norwitz
J dot Russell Noseworthy
Michelangelo Nottoli
Vsevolod Novikov
Jaroslaw Nozderko
Otis Nyandoro
Ken OBrien'
Frank ODwyer'
David O'Farrell'
David O'Farrell'
Philippe O'Reilly'
Carlos O'Ryan(DOC)'
David-A O'Brien
Pierre Oberson
Jerry Odenwelder
Sree Oggu
Sean Ogle
Shaun Ohagan
David Ohlemacher
Rick Ohnemus
Diethard Ohr
Thaddeus Olczyk
Eider Oliveira
Brian C. Olson
Andy Olson
Brian Olson
Steve Olson
Bob Olson
Anders Olsson
Jeff Olszewski
Yev Omenzel
Gul Onural
<table>
<thead>
<tr>
<th>TAO and ACE Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Norbert Rapp</td>
</tr>
<tr>
<td>Brian Raven</td>
</tr>
<tr>
<td>Narendra Ravi</td>
</tr>
<tr>
<td>Michael Ravits</td>
</tr>
<tr>
<td>Sarooj Reddy</td>
</tr>
<tr>
<td>Chris Reed</td>
</tr>
<tr>
<td>Kevin Regan</td>
</tr>
<tr>
<td>Shmulik Regev</td>
</tr>
<tr>
<td>Karen L. Regner</td>
</tr>
<tr>
<td>George Reid</td>
</tr>
<tr>
<td>Andrew Reid</td>
</tr>
<tr>
<td>Wilfried Reinoehl</td>
</tr>
<tr>
<td>Jonathan Reis</td>
</tr>
<tr>
<td>Richard Reitmeyer</td>
</tr>
<tr>
<td>Margaret Reitz</td>
</tr>
<tr>
<td>Guillaume Renaud</td>
</tr>
<tr>
<td>Pavel Repin</td>
</tr>
<tr>
<td>George Reynolds</td>
</tr>
<tr>
<td>Sam Rhine</td>
</tr>
<tr>
<td>Orlando Ribeiro</td>
</tr>
<tr>
<td>Michael Rice</td>
</tr>
<tr>
<td>Jeff Richard</td>
</tr>
<tr>
<td>Hans Ridder</td>
</tr>
<tr>
<td>Alexander Rieger</td>
</tr>
<tr>
<td>Bryon Rigg</td>
</tr>
<tr>
<td>Michael Rinne</td>
</tr>
<tr>
<td>Francois Rioux</td>
</tr>
<tr>
<td>Jean-Francois Ripouteau</td>
</tr>
<tr>
<td>James Risinger</td>
</tr>
<tr>
<td>Steve Ritter</td>
</tr>
<tr>
<td>Loren Rittle</td>
</tr>
<tr>
<td>Ahmed Riza</td>
</tr>
<tr>
<td>Bill Rizzi</td>
</tr>
<tr>
<td>Bjorn Roald</td>
</tr>
<tr>
<td>Gerwin Robert</td>
</tr>
<tr>
<td>John D. Robertson</td>
</tr>
<tr>
<td>Rick Robinson</td>
</tr>
<tr>
<td>Jim Robinson</td>
</tr>
<tr>
<td>David Robison</td>
</tr>
<tr>
<td>John Rodgers</td>
</tr>
<tr>
<td>Craig Rodrigues</td>
</tr>
<tr>
<td>Oscar Rodriguez</td>
</tr>
<tr>
<td>Jim Rogers</td>
</tr>
<tr>
<td>Thomas Rohner</td>
</tr>
<tr>
<td>Hans Rohnert</td>
</tr>
<tr>
<td>Keith Rohrer</td>
</tr>
</tbody>
</table>
TAO and ACE Contributors

Renjie Tang
Alan Tang
Marc Tardif
Chris Tarr
Mason Taube
Charles Taurines
John R. Taylor
Carlton Teel
Anders W. Tell
Boris Temkin
Ty Tenait
Victor Terber
Oleg Terletsky
Andreas Terstegge
Gautam Thaker
Chander P. Thareja
Brodie Thiesfield
Jerry Thomas
Michael Thomas
Greg Thompson
Matt Thompson
Edward A Thompson

Rob Thornton (DOC)
Keith Thornton
Arjun Thounaojam

Bryan Thrall (DOC)
Gergely Timar
Steven Tine
Sathish Tiptur
Dror Tirosch
Andreas Tobler
Jarek Tomaszewski
Bill Tonseth
Jason Topaz
Audun Tornquist
Levente Torok
Stephen Torri
Antonio Tortorici
Alain Totouom

Steve Totten (OCI)
Bill Tovrea
Matthew Townsend
W Craig Trader
Corey Trager
Roger Tragin
Bruce Trask

Karl Tredwell
Joachim Tremouroux
Yung Trinh

Daniel Troessler (OCI)
Gabriele Trombetti (DOC)

David Trumble
David Trusty
Stanislaw Trytek
John Tucker
Petr Tuma

Emre Turkay (DOC)
Andreas Ueltschi
Satoshi Ueno
Stefan Ulrich
Hagen Ulrich
Cristian Ungureanu
Serkan Unsal
Erik Urdang
Hans Utz
Chris Uzdavinis
Sankaranarayanan K. V
Aaron Valdivia
Antti Valtokari
Matthieu Vansteene
Mahesh Varadarajan
George Varsamis
Eric Vaughan

Mahesh Vedantam (OCI)

Christophe Vedel
Srikanth Vedire
Christian Veleba
Momchil Velikov
Bryan Van de Ven
Chandra Venkatapathy
Thomas Venturella
Nicoletta Viale
Simone Viani
Benoit Viala
Ephraim Vider
Paulo Breda Vieira
Derek Viljoen
Edgar Villanueva
Alberto Villarica
Alexander Villatora
Alex Villazon

Adamo, Vince
Steve Vinoski
Ram Vishnuvaj jala
Bob Vistica
Mike Vitalo
Ulrich Voigt
William R Volz
Max Voronoy
Gerhard Voss
Andrew Voumard
Steve Vranyes
Hai Vu
Wayne Vucenic
Jeff W
Mathias Waack
Jonathan Wackley
Andreas Wagner
Tom Wagner
Jerome Waihel
Brian Wallis
Marc Wahrave
Brian Waltersdorf
Byron Walton
Brad Walton
Jia Wan
David Yongqiang Wang

Nanbor Wang (DOC)
Herbert Wang
Phil Y. Wang
Xiaowen Wang
Judy Ward
Terry Ware
Doug Warner
Troy Warner
Steve Warwick
Craig Watcham
Garth Watney
Kent Watsen
John Weald
Peter Weat
Clarence M. Weaver
Rudolf Weber
Siegurd Weber
Roger Weeks
Mark Weel
References


Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.


References


1. See page xxxi in the Preface for instructions on obtaining OMG documents.


References


Index

Symbols

_add_ref() operation 322, 363, 1239
_cxx_register() operation 1189
_get_policy() operation 268
_id() operation 104, 106–107, 109, 157, 313, 391, 831–832
_make_resources() operation 597
_narrow() operation 367
_refcount_value() operation 322
_remove_ref() operation 322, 363, 1239, 1256
_retn() operation 1247
_set_policy_overrides() operation 170
Unchecked_narrow() operation 339–340, 392
_validate_connection() operation 174, 213, 228

A

abstract factory pattern 15
abstract keyword
  value types 330

abstract value type 328–330
accept_svc_handler() operation 423
acceptor 620, 689
acceptor pattern 15
access identity 1070
ACE 737
  service configurator 11, 591–592, 594–606, 608–616, 621, 682, 690, 713, 739, 748,
  763, 766, 769
  _make_resources() operation 597
ACE_Dynamic_Service class 614
ACE_FACTORY_DECLARE macro 611
ACE_FACTORY_DEFINE macro 611
ACE_SERVICE_ALLOCATOR 609
ACE_Service_Config class
  close() operation 601
  open() operation 599–600, 621, 630–631, 682–683
  process_directive() operation 602
  reconfigure() operation 602, 616
ACE_Service_Object class 596, 603, 605, 607, 609
ACE_STATIC_SVC_DECLARE macro 611
ACE_STATIC_SVC_DECLARE_EXPORT macro 611
ACE_STATIC_SVC_DEFINE macro 611
ACE_STATIC_SVC_REQUIRE macro 612
ACE_SVC_NAME macro 610, 612
ACE_SVC_OBJ_T macro 610
Client_Strategy_Factory configuration 595
  commanding 601
  control options 597
DELETE_OBJ macro 610
DELETE_THIS macro 610
destructor 608
dynamic directive 596, 605, 608, 612
example
  dynamic service 612
  static service 613
factories 592
finalizer 608
fini() operation 608, 610
framework 598
helper macros 610
info() operation 614
init() operation 608
instance() operation 615
loading service objects 598
obtaining services 614
options 595
ORB initialization options 96, 99, 597, 599, 617–618, 625, 1053
  ORBServiceConfigLoggerKey 598
ORBSCvcConf 598
  ORBSCvcConfDirective 598
process_directive() operation 616
remove directive 602, 606, 608
Resource_Factory configuration 595–596, 605
resume directive 602–603, 606, 614
Server_Strategy_Factory configuration 595
service finalization 608
service initialization 608
service manager 592, 615
service objects 603, 606
service state 614
static directive 595, 605, 608–609, 615, 739, 748
suspend directive 602–603, 606, 614
XML 592, 603

service manager 601–603

**ACE classes**

ACE_Acceptor 422–424
ACE_Addr 437, 481, 525, 909
ACE_Allocator 691
ACE_Data_Block 693–694
ACE_Dynamic_Service 492, 614–615
ACE_Dynamic_Service_Base 614
ACE_Event_Handler 492, 614–615
ACE_IPC_SAP 525
ACE_Log_Msg 601
ACE_MEM_Acceptor 416
ACE_MEM_Connector 417
ACE_Message_Block 693, 695, 719
ACE_Mutex 1239
ACE_QtReactor 707–708
ACE_Select_Reactor 570, 727
ACE_Service_Config 491–492, 598–602, 616, 621, 630–631, 677, 682–683
close() operation 601
open() operation 599–600, 621, 630–631, 682–683
process_directive() operation 602
reconfigure() operation 602, 616
ACE_Service_Object 465, 486–488, 493, 596, 603, 605–607, 609, 612, 691, 740–
741, 764
ACE_Service_Object ACE_Service_Object class 610, 612–613
ACE_Service_Object_Exterminator 487
ACE_Service_Repository 489
ACE_Service_Type 610
ACE_Shared_Continuation 606–607
ACE_Singleton 336
ACE.SOCK_Acceptor 416
ACE.SOCK_Connector 417
ACE_Static_Svc_Descriptor 487–488, 491, 609, 611
ACE_Strategy_Acceptor 416, 419, 421–424, 431, 446
ACE_Strategy_Connector 417, 443–444, 446–447, 449, 454
ACE_Svc_Handler 464, 466, 505
ACE_Task 465
ACE_Task_Base 565, 811–812, 1254, 1257
ACE_TP_Reactor 570
ACE_UNIX_Addr 420, 430, 433, 435–436, 438, 458, 469, 472, 474
ACE_XtReactor 709

ACE macros
__ACE_INLINE__ 85
ACE_ANY_EXCEPTION 115, 117, 124, 127–131
ACE_CATCH 114–121, 124, 126–131, 137
ACE_CATCHALL 115, 117, 119, 124
ACE_CATCHANY 115, 117, 119, 124, 127–131
ACE_CHECK 115, 117–119, 122–124, 126–131, 464
ACE_CHECK_RETURN 115, 117, 122–124, 126–127, 464
ACE_COMPILE_TIMEPROBES 1383
ACE_DECLARE_NEW_CORBA_ENV 113, 124–126, 128–130, 710
ACE_DECLARE_NEW_ENV 113
ACE_DEFAULT_LOGGER_KEY 600, 677
ACE_DEFAULT_MAX_SOCKET_BUFSIZE 675, 680
ACE_ENTRY 114, 117, 119–121, 124, 126–131, 708, 710
ACE_ENV_ARG_DECL 93–94, 113, 442, 477–478, 482–483
ACE_ENV_ARG_DECL_NOT_USED 113
ACE_ENV_ARG_DECL_WITH_DEFAULTS 113
ACE_ENV_ARG_PARAMETER 114, 116, 118, 126, 128–130
ACE_ENV_BKWD_COMPAT 112, 1380
ACE_ENV_SINGLE_ARGDECL 114, 119–121, 123, 442–443, 463–464, 478, 485
ACE_ENV_SINGLE_ARGDECL_NOT_USED 114
ACE_ENV_SINGLE_ARGDECL_WITH_DEFAULTS 114
ACE_ENV_SINGLE_ARG_PARAMETER 114, 116–118, 123–130, 511
ACE_FACTORY_DECLARE 487, 611–613, 702
ACE_FACTORY_DEFINE 488, 611–613, 702
ACE_HAS_EXCEPTIONS 93–94, 112, 1379
ACE_HAS_FL 1384
ACE_HAS_NO_THROW_SPEC 116
ACE_HAS_QT 1384
ACE_HAS_TK 1384
ACE_HAS_XML_SVC_CONF 603
ACE_HAS_XT 1383–1384
ACE_IPC_SAP 505
ACE_LACKS_RTTI 1380
ACE_MAX_DGRAM_SIZE 387
ACE_NDEBUG 1381
ACE_NO_INLINE 1380
ACE_PRINT_EXCEPTION 115, 117, 119–121, 124, 126–131
ACE_RE_THROW 115, 121
ACE_SERVICE_ALLOCATOR 609
ACE_STATIC_SVC_DECLARE 487, 491, 611
ACE_STATIC_SVC_DECLARE_EXPORT 491, 611, 613, 702
ACE_STATIC_SVC_DEFINE 611, 613, 702
ACE_STATIC_SVC_REQUIRE 491, 612–613, 702
ACE_SVC_NAME 610, 612–613
ACE_SVC_OBJ_T 610, 613
ACE_TCHAR 610
ACE_THROW_RETURN 114, 122–123
ACE_THROW_SPEC 25, 91, 93–94, 116, 325, 346, 361
ACE_TRY 113–122, 124–131, 137, 708, 710, 1380
ACE_TRY_CHECK 115–119, 124–127, 129–131, 708, 710
ACE_TRY_CHECK_EX 115, 125–127, 131
ACE_TRY_ENV 113–115, 117–118, 122, 1377, 1380
ACE_TRY_EX 113–115, 120, 125–127, 131
ACE_TRY_NEW_ENV 113, 116–118
ACE_TRY_THROW 114–115, 120–122, 131
ACE_TRY_THROW_EX 114–115, 120–122, 131
DELETE_OBJ 488, 610, 613
A

DELETE_THIS 488, 610, 613

ACE service configurator
See ACE, service configurator

ACE_Acceptor class 422–423
ACE_Addr class 437, 481, 525, 909
ACE_Allocator class 691
ACE_Data_Block class 693–694
ACE_Dynamic_Service class 492, 614–615
ACE_Dynamic_Service_Base class 614
ACE_Event_Handler class 465, 467, 472–473, 510, 545–546, 606–607
ACE_HAS_XML_SVC_CONF macro 603
ACE_IPC_SAP class 525
ACE_Log_Msg class 601
ACE_MEM_Acceptor class 416
ACE_MEM_Connector class 417
ACE_Message_Block class 693, 695, 719
ACE_Mutex class 1239
ACE_QtReactor class 707–708
ACE_ROOT environment variable 22, 84, 88, 97, 1333, 1349, 1352, 1365, 1392
ACE_Select_Reactor class 570, 727
ACE_Service_Config class 491–492, 598–602, 616, 621, 630–631, 677, 682–683
ACE_Service_Object class 465, 486–488, 493, 596, 603, 605–607, 609, 612, 691, 740–741, 764
ACE_Service_Object_Exterminator class 487
ACE_Service_Repository class 489
ACE_Service_Type class 610
ACE_Shared_Object class 606–607
ACE_Singleton class 336
ACE.SOCK_Acceptor class 416
ACE.SOCK_Connector class 417
ACE_Static_Svc_Descriptor class 487–488, 491, 609, 611
ACE_Strategy_Acceptor class 416, 419, 422–424, 431
ACE_Strategy_Connector class 417, 443, 446, 449
ACE_Svc_Handler class 464, 466, 505
ACE_Task class 465
ACE_Task_Base class 565, 811–812, 1254, 1257
ACE_TP_Reactor class 570
ACE_UNIX_Addr class 420, 430, 433, 436, 438, 458, 469, 472, 474
ACE_XtReactor class 709
activate() operation 26, 298, 567, 610, 742, 794, 812–813, 992, 1016, 1131, 1253–1254, 1257, 1259, 1279
activate_object() operation 26
activate_object_with_id() operation 157, 391
activate_object_with_id_and_priority() operation 271
activate_object_with_priority() operation 271
activate_svc_handler() operation 423
activation 157
activation-per-AMI-call strategy 156–157
active demultiplexing 10, 15, 572, 586, 627–628, 745, 752, 754, 757, 760, 762
active object map 157, 356, 744, 747, 751, 761
parameters 747
active object pattern 15
adapter pattern 15
ADAPTIVE Communication Environment (ACE)
See ACE
add_client_request_interceptor() operation 292
add_constraints() operation 1002
add_filter() operation 1002
add_ior_component() operation 308
add_ior_interceptor() operation 307–308
add_server_request_interceptor() operation 297
add_transport_to_cache() operation 472
addr_to_string() operation 475
Advanced CORBA Programming Using TAO course xxxvi
Advanced CORBA Programming with C++ book xxvii, xxxii, xxxvi, 8, 21, 102–103, 143, 171, 200, 332, 408, 693, 791–792, 827, 955, 1139, 1465
advanced resource factory 380–381, 568, 690, 692, 694, 697, 705, 710–712, 734
options 730
alert protocol 1097
AMH
See asynchronous method handling (AMH)
amh_response_handler_allocator() operation 704
AMI
See asynchronous method invocation (AMI)
ami build flag 1378, 1385
ami_response_handler_allocator() operation 704
ANY_EVENT type 984, 990
apply_project option 38
assembly
CORBA Component Model (CCM) 1286, 1289–1290, 1312–1317, 1320–1323, 1325–1328
assembly descriptor
CORBA Component Model (CCM) 1215, 1217

assembly implementation
CORBA Component Model (CCM) 1301

asymmetric encryption 1089
asymmetric protocol 393–394
asynchronous invocations 234
asynchronous method handling (AMH) 135, 185–209, 533, 704, 1465
  advantages 186
  disadvantages 187
  example 188, 191
  response handler 188, 190
  servant 188
  skeleton 188
  using with
    collocation 202
    CORBA::Current objects 201
    portable interceptors 200
    reference counted servants 201

asynchronous method invocation (AMI) 139, 142, 186, 332, 531, 585, 704, 1467
  activation-per-AMI-call strategy 156–157
  associating replies with requests 156
  callback 140–141, 160, 162, 190
  drawbacks 142
  example 160
  exception replies 148
  ExceptionHolder class 142, 145–146, 148
  IDL compiler 142
  local interfaces 354
  processing sendc_ operation 158
  reply handler 147, 150
    exception 147, 155
    non-exception 147, 151
  request delivery 167
  sendc_prefix 141, 143, 158
  servant-per-AMI-call strategy 156–157
  server-differentiated-reply strategy 156–157
  synchronous IDL operation 146

attribute
attribute keyword
CORBA Component Model (CCM) 1223

audio/video streaming service 13, 783
ORB service libraries 783

auditing 1072
identity 1070

automake build type 36
AV streams
See audio/video streaming service

BAD_INV_ORDER exception 135, 541–542, 545
BAD_PARAM exception 134, 309
BAD_QOS exception 969
base option 37
basic filter builder
real-time event service (RTES) 887
basic object adapter (BOA) 10
begin_scheduling_segment() operation 236, 238, 273, 275
Berg, Daniel 1466
bi-directional GIOP 393
BidirectionalPolicyValue 181
See also general inter-ORB protocol (GIOP), bi-directional
bind() operation 366
binding
direct 1138
indirect 1138
bindings 798, 814, 816, 1084–1085, 1091
explicit 10, 213, 228
indirect 366, 1151
bit-mask filters 889
real-time event service (RTES) 889
blocking strategy 719
BlockingPolicy property 966, 1026
bmake build type 36
BOA
See basic object adapter (BOA)

**Bolton, Fintan** xxvii, xxxii, 1465  
**Boost** 1345, 1347–1349, 1351–1352  
  building 1347, 1351  
**Boost Jam** 1347–1348, 1351–1352  
**BOOST_INCLUDE environment variable** 1352  
**BOOST_LIB environment variable** 1352  
**BOOST_ROOT environment variable** 1352  
**Borland** 19, 27, 29–30, 33, 38, 51, 69  
**Borland Make** 29  
**boxed value types** 330  
**bridge pattern** 15  
**Brunschw, Darrell** 1465  
**BUFFER_MESSAGE_COUNT** 178–179  
**BUFFER_TIMEOUT** 178  
**buffering**  
  oneway 177  
  requests 214  
**BufferingConstraint policy** 177–179  
**build flags** 335, 341  
  ami 1378, 1385  
  corba_messaging 262, 1378, 1385  
  debug 1378, 1381  
  exceptions 1377, 1379  
  fast 1378, 1381  
  fl_reactor 60, 711, 1378, 1384  
  gtk_reactor 1378, 1384  
  include_env 112, 1377, 1380  
  inline 49–50, 52, 54–55, 62, 65, 85, 89, 1360, 1378, 1380  
  interceptors 1378, 1385  
  minimum_corba 182–183, 262, 282, 1378, 1383, 1385–1386  
  optimize 1378, 1381  
  pipes 1379, 1386  
  probe 1378, 1383  
  profile 1378, 1383  
  purify 1378, 1382  
  qt_reactor 1378, 1384  
  quantify 1071, 1378, 1382  
  repo 1381  
  rt_corba 262, 1378, 1385  
  rtti 1377, 1380  
  rwho 1378, 1386
building

ACE_ROOT/ace/config.h 86, 112, 603, 708–709, 1377
ACE_ROOT/bin/generate_export_file.pl 97, 611
ACE_ROOT/bin/svcconf-convert.pl 604
ACE_ROOT/include/makeinclude/platform_macros.GNU 1377
ACE and TAO 17, 1389
  from source code distribution 1389
  on UNIX 1389
  UNIX 1391–1400
  Visual C++
    build libraries 1350, 1353
  with Borland C++ 1390, 1409–1413
    configuring source code 1411
    setting up environment 1410
    verify build 1413
  with Visual C++ 1389, 1401
    build libraries 1405
    configure source code 1403
    custom version 1401
    setting up environment 1403
    verify build 1405

cross compilation
MakeProjectCreator (MPC) 19, 21, 37, 41, 76, 1392, 1395
messaging 1385
security libraries 1106
  on UNIX 1107
C

on Windows 1108

**built-in topic**

- data-centric publish-subscribe (DCPS) 1179, 1203–1206, 1209

**Buschmann, Frank** 16, 404, 563, 582, 1466–1467

**byte-order flag** 379, 725

---

**C**

**C++ language mapping** 5, 12, 1442

- CORBA compliance 1442
- exceptions 1377, 1443
- modules 1443
  
  *See also interface definition language (IDL), C++ mapping*

**C++ Network Programming book**

- Volume 1 (C++NPv1) 16, 404, 525, 528, 534–535, 1468
- Volume 2 (C++NPv2) 16, 404, 540, 555, 568, 570, 592, 1468

**cache_maximum() operation** 696–697

**caching**

- cache_maximum() operation 696–697
- createCached_connection_lock() operation 696
- create_purging_strategy() operation 696
- locked_transport_cache() operation 696
- management 696
- max_muxed_connections() operation 696–697
- purge_percentage() operation 696

**Cadena** 1289

**cancel() operation** 236

**catior utility** 309

**cbx build type** 36

**CCM**

*See CORBA Component Model (CCM)*

- **ccm_activate() operation** 1253–1254, 1257, 1259–1260, 1268, 1271, 1279–1280
- **ccm_passivate() operation** 1253, 1259–1260, 1268, 1271
- **ccm_remove() operation** 1253, 1259, 1268, 1271

**CDR**

*See common data representation (CDR)*

- **CECConsumerControl option** 849, 853–855, 861, 868
- **CECConsumerControlPeriod option** 849, 854–855, 861, 867–868
- **CECConsumerControlTimeout option** 849–850
CECDispatching option 847–848, 851, 856–857
CECDispatchingThreads option 847–848, 856–857
CECProxyConsumerCollection option 850, 858–859
CECProxyConsumerLock option 848, 860
CECProxyDisconnectRetries option 849
CECProxySupplierCollection option 850, 862–863
CECProxySupplierLock option 848, 864
CECReactivePullingPeriod option 852, 865
CECSupplierControl option 849, 866–867
CECSupplierControlPeriod option 849, 867
CECSupplierControlTimeout option 849–850
CECUseORBId option 852, 869
certificates 1091, 1099
authority 1091, 1100
commands summary 1106
creating requests 1101
issuing 1103
multiple authorities 1112
self-signed 1092
signing requests 1103
character set
definition 703
CIAO
See Component-Integrated ACE ORB (CIAO)
ciao_postactivate() operation 1253–1254, 1260, 1268, 1271
ciao_preactivate() operation 1253–1254, 1260, 1268, 1271
CIAO_ROOT environment variable 1290, 1331–1332, 1345, 1349–1350, 1352–1353, 1357, 1365
CIDL
See component implementation definition language (CIDL)
building 1345–1353
desc-file-regex option 1364
desc-file-suffix option 1364
exec-export-include option 1363
exec-export-macro option 1363
exec-hdr-file-regex option 1363
exec-hdr-file-suffix option 1363
exec-src-file-regex option 1363
exec-src-file-suffix option 1363
gen-exec-impl option 1362
help option 1359–1360
help-html option 1360
include path 1336, 1359
input files 1232
lem-file-regex option 1362
lem-file-suffix option 1362
lem-force-all option 1362
options 1359–1364
output files 1232–1233, 1235, 1237–1239, 1241, 1249, 1251, 1356–1357, 1362, 1364
preprocessing 1359
preprocess-only option 1359
reference 1356–1364
starter code 1362
suppress-register-factory option 1360
svnt-export-include option 1361
svnt-export-macro option 1361
svnt-hdr-file-regex option 1361
svnt-hdr-file-suffix option 1360
svnt-src-file-regex option 1361
svnt-src-file-suffix option 1361
trace-semantic-actions option 1360
cidlc program 1356–1359
CIF
See component implementation framework (CIF)
ciphertext 1088
client interceptors 282–283, 289, 301
client role 529
cached connection strategy 715
connect strategy 768, 770
create_ft_service_retention_id_lock() operation 764
interface definition 764
multiplexing strategies 765
options 769
ORBClientConnectionHandler 768
ORBConnectStrategy 573, 580, 769–770, 1115
ORBProfileLock 558, 765, 771
ORBTransportMuxStrategy 561, 574, 580, 766, 772
profile locking 764, 771
TAO_Wait_Strategy class 766, 768
transport multiplexing 765
wait strategy 766, 773


ClientInterceptor class 288
client-propagated priority model 266, 268
ClientRequestInfo interface 285
get_effective_component() operation 307

close_connection() operation 470, 504
CLOSE_WAIT socket state 584
code set 703
code set configuration 703–704
definition 703
code set translator 703

codec (coder/decoder) 300
codec_factory() operation 300, 304
CodecFactory reference 300–301
codepoints
diffserv 252

collocation 9
asynchronous method handling (AMH) 202
event channel 834
notification channel
example 1015
objects 531, 623, 628, 836
optimization 586–588
real-time collocation resolver 265, 587, 640

COMM_FAILURE exception 103, 107, 117, 133
common data representation (CDR) 372–373, 593, 695, 1182
conversion allocators 693

common secure interoperability (CSI) packages 1066, 1075, 1078–1079
CSI 0 1075, 1078
CSI 1 1066, 1075, 1078
CSI 2 1066, 1075, 1078

complete option 39
compliance
See CORBA compliance

component 1214
collaboration 1215
CORBA Component Model (CCM) 1213, 1223
definition 1214
provided interface 1215
uses interface 1215
component connection 1216
component container
component deployment plan 1290, 1323–1324, 1327–1330, 1344
component domain descriptor 1290, 1323–1324, 1328–1330
component executor 1231, 1233, 1292
component home 1225–1229, 1231, 1260–1261, 1264, 1275, 1356, 1370
component implementation 1215
component implementation definition language (CIDL) 1215, 1227–1229, 1231, 1249, 1260, 1266, 1272, 1283, 1336, 1356–1360
cidlc program 1356–1359
compiler
See CIDL compiler
preprocessing options 1359
component implementation descriptor 1289, 1293, 1301, 1303–1304, 1308, 1311–1315, 1320–1321, 1324, 1327–1328
component implementation framework (CIF) 1215, 1227
component keyword
CORBA Component Model (CCM) 1223, 1227
component package descriptor 1289, 1293, 1303–1305, 1308, 1311–1312, 1314, 1316, 1320–1321, 1325–1326
component property 1215
Component-Integrated ACE ORB (CIAO) xxviii, 1213–1370
building 1345–1353
example 1219
ExecutionManager program 1343, 1354–1356
NodeApplication program 1343, 1345, 1354–1355
NodeManager program 1329, 1343, 1345, 1354–1355
RepositoryManager program 1343, 1354, 1356
composite pattern 1286
composition
CORBA Component Model (CCM) 1215, 1228–1230, 1249, 1260, 1266, 1272, 1275, 1283, 1321, 1356, 1362
composition keyword
CORBA Component Model (CCM) 1231

**concurrency**  xxxvi, 4, 11–12, 15, 591, 739, 741
  behavior  753, 758
  server  741
  strategy  742

**concurrency control service**  4, 13, 784
  ORB service libraries  784

**concurrency models**  387, 392, 427, 554–565, 620
  reactive  427, 554–559, 564
  thread-per-connection  387, 392, 426, 471, 554, 559–562, 743, 759, 838
  thread-pool  554, 562–568, 570, 586
  wait-on-leader-follower  774

**concurrency-strategy template**  426, 451

**condition**
  data-centric publish-subscribe (DCPS) 1180


**configuration**
  source code
    See also building

**conjunction groups**  890

**connect strategies**  572–573, 768
  blocking  573
  connect-strategy template  450
  leader-follower  572
  reactive  573

**connect_push_consumer() operation**  837, 881, 886, 901
**connect_push_supplier() operation**  829, 876, 901
**connect_sequence_push_consumer() operation**  1014
**connect_sequence_push_supplier() operation**  1009
**connect_structured_push_consumer() operation**  991
**connect_structured_push_supplier() operation**  985

**connecting and disconnecting**
  real-time event service (RTES)  881

**connection handler**  689

**connection timeout**  174

**connection_handler_i() operation**  510
**ConnectionReliability property**  962, 966, 970, 976

**connector**  689
**connector pattern**  15

**consolidate_message() operation**  508

**consumer**
CORBA Component Model (CCM) 1214–1215, 1221

**consumer and supplier control options** 848, 918
- CECConsumerControl 849
- CECConsumerControlPeriod 849
- CECConsumerControlTimeout 849–850
- CECSupplierControl 849
- CECSupplierControlPeriod 849
- CECSupplierControlTimeout 849–850
- ECCConsumerControl 918, 922, 943–948
- ECCConsumerControlPeriod 918, 923
- ECOBserver 928
- ECSupplierControl 918, 936
- ECSupplierControlPeriod 918, 937

**consumer proxies** 872

**ConsumerAdmin interface** 950, 958, 989–990, 996, 1003, 1013

**consumes keyword**
- CORBA Component Model (CCM) 1214, 1225, 1227, 1299

**context_data field** 290

**context_id field** 290

**CORBA**
- architecture 8, 214
- audio/video streaming service 13
- client role 529
- compliance xxvii, 7, 12, 14, 1435–1447
  - C++ language mapping 1442
  - CORBA Component Model (CCM) 1445
  - core 1437
  - data distribution service (DDS) 1445
  - DII COE 1446
  - interoperability 1437
  - interworking 1438
  - minimum CORBA 1439
  - naming service 1443
  - notification service 1444
  - quality of service (QoS) 1439
  - real-time CORBA 1441
  - security service 1445
  - value types 330
- concurrency control service 13
- CORBA::Any 300, 827
- CORBA::Boolean 314
- CORBA::OctetSeq 304
core 1437
   specification 140
data distribution service (DDS) 13
environment parameter 83
   exceptions 102, 110, 112–115, 117, 137, 1380
event service 13
gateway 905
interface repository 13
interoperability 1437
interworking 1438
life cycle service 13
load balancing service 13
log service 13
messaging
   See messaging
minimum 1383
naming service 13
notification service 13
object 9, 133, 627, 681, 791, 836
ORBPolicyManager interface 230
PolicyCurrent interface 230
PolicyList interface 224, 226
property service 13
quality of service (QoS) 1439
real-time 64, 135, 400, 474, 554, 561, 565, 1467
   See also real-time CORBA
security service 13
server role 529, 531
services 13
system exceptions 103
time service 13
trading service 13
VoidData native type 237

CORBA component descriptor 1289, 1292, 1297, 1301–1304, 1306, 1308–1314, 1316–1317, 1321, 1364

CORBA Component Model (CCM) 1213
   assembly 1286, 1289–1290, 1312–1317, 1320–1323, 1325–1328
   assembly descriptor 1215, 1217
   assembly implementation 1301
   attribute 1219, 1223, 1237, 1241–1242, 1244–1245, 1252, 1257, 1289, 1297–1299,
   1301, 1304, 1313–1314, 1316–1317, 1320, 1325–1326, 1330, 1344
   definition 1215
attribute keyword 1223
component 1213–1214, 1223
  collaboration 1215
definition 1214
  provided interface 1215
  uses interface 1215
component connection 1216
component container 1216–1217, 1227, 1233, 1239, 1249, 1251–1254, 1258–1261,
  1263–1264, 1266–1268, 1270–1271, 1285, 1322
component deployment plan 1290, 1323–1324, 1327–1330, 1344
component domain descriptor 1290, 1323–1324, 1328–1330
component executor 1231, 1233, 1292
component home 1225–1229, 1231, 1260–1261, 1264, 1275, 1356, 1370
component implementation 1215
component implementation definition language (CIDL) 1215, 1227–1229, 1231,
  1249, 1260, 1266, 1272, 1283, 1336, 1356–1360
component implementation descriptor 1289, 1293, 1301, 1303–1304, 1308, 1311–
  1315, 1320–1321, 1324, 1327–1328
component implementation framework (CIF) 1215, 1227
component keyword 1223, 1227
component package descriptor 1289, 1293, 1303–1305, 1308, 1311–1312, 1314,
  1316, 1320–1321, 1325–1326
component property 1215
component server 1216–1217, 1225, 1287–1288, 1290, 1293, 1302–1304, 1323–
  1324, 1329–1330, 1343–1344
composition 1215, 1228–1230, 1249, 1260, 1266, 1272, 1275, 1283, 1321, 1356,
  1362
definition 1228
composition keyword 1231
consumer 1214, 1221
definition 1215
consumes keyword 1214, 1225, 1227, 1299
CORBA component descriptor 1289, 1292, 1297, 1301–1304, 1306, 1308–1314,
  1316–1317, 1321, 1364
deployment 1213, 1216, 1218, 1224–1226, 1267, 1274–1275, 1285–1291, 1312,
  1315, 1318, 1323–1325, 1329–1330, 1343–1345, 1356, 1364, 1370
deployment descriptor 1217, 1289–1292, 1322
definition 1288
deployment node 1323
deployments keyword 1224, 1227, 1299
entity composition 1229
entity keyword 1231
event 1214–1215, 1217, 1219–1220, 1222–1227, 1244, 1248, 1252, 1254, 1266–1268, 1270, 1307
event consumer 1227, 1293, 1297, 1299, 1306–1307, 1340
event emitter 1224, 1297, 1299
event publisher 1293, 1297, 1299–1300
event sink 1216
   definition 1215
event source 1216
   definition 1215
event type 1215, 1219, 1221–1222, 1224–1225, 1227, 1244, 1300, 1307, 1335, 1360
event type factory 1222, 1360
eventtype keyword 1222, 1227
example 1219
   definition 1238
executor keyword 1231
   definition 1215
home executor 1229–1231, 1233, 1260–1264, 1267, 1272, 1283, 1292
home keyword 1224, 1227
implementation artifact descriptor 1289, 1292–1297, 1302, 1304, 1306, 1308–1309, 1311
implements keyword 1231
manages keyword 1224, 1227, 1231
monolithic implementation 1293, 1301, 1303
multiplex receptacle 1274–1277, 1297, 1299, 1310–1311
node map 1290, 1323–1324, 1327, 1329, 1344, 1354–1355
package configuration descriptor 1289, 1323–1327, 1330
port 1215–1216, 1224, 1265, 1291, 1293, 1297, 1299–1301, 1307, 1310–1311, 1317, 1319, 1329, 1354
   definition 1215
process composition 1229
process keyword 1231
programming model 1216
provides keyword 1214, 1224, 1227, 1299
definition 1215
publishes keyword 1214, 1224, 1227, 1299
receptacle 1216, 1219, 1265, 1273–1277, 1281–1283, 1307, 1317, 1319
definition 1215
service composition 1229
services keyword 1231
session composition 1228
session keyword 1231
simplex receptacle 1275–1276, 1297, 1299, 1306–1307
subscriber 1224
supports keyword 1298, 1370
top-level package descriptor 1289, 1323–1324, 1326–1327, 1330, 1344, 1356
uses keyword 1225–1227, 1269, 1299
uses multiple keyword 1226–1227, 1273–1275, 1299
CORBA Programming with C++ course xxxv
CORBA Programming with Java course xxxv
corba_messaging build flag 262, 1378, 1385
corbaloc
object URL
example 801
corbaloc object URL 365–366, 786–787, 800–803, 808, 815
example 802
iiop protocol 801
rir protocol 801
corbaname object URL 720, 803–804, 806–807, 810, 814, 1443
example 804
CosEvent_Service program 827, 834
command line options 845–846
CoSMIC 1288
create() operation 1225, 1263–1264
create_cached_connection_lock() operation 696–697
create_channel() operation 982, 1016
create_corba_object_lock() operation 705
create_filter() operation 1001
create_flushing_strategy() operation 695
create_for_unmarshal() operation 324
create_ft_service_retention_id_lock() operation 764
create_if_strategy() operation 692
create_mutex() operation 223
create_named_mutex() operation 259
create_participant() operation 1187
create_POA() operation 171, 215, 219, 224
create_policy() operation 168–169, 172, 174, 176, 179, 182, 1121–1124
create_priority_banded_connection_policy() operation 228
create_purging_strategy() operation 696–697
create_reference_with_id_and_priority() operation 271
create_reference_with_priority() operation 271
create_resource_manager() operation 240, 242
creation-strategy template 424, 449
cross compilation
  See building, cross compilation

CSI
  See common secure interoperability (CSI) packages
CSI ECMA protocol 1075, 1079
Current interface 157, 221, 268, 311, 355, 1125
custom factory 725
custom interoperable object reference (IOR) parsers 699
customer support
  See support
customizing
  access to services 782
  ACE and TAO builds
    See building
event structure 885
  filters 894

D

DAnCE
  See Deployment And Configuration Engine (DAnCE)
data distribution service (DDS) 13, 1445
  compared to event service 1176
  compared to notification service 1176
  compared to real-time event service (RTES) 1176
data local reconstruction layer (DLRL) 1175
data-centric publish-subscribe (DCPS) 1175
  overview 1176–1180
  example 1183–1186, 1194–1195, 1205
options
  DCPSChunks  1206–1207
  DCPSDebugLevel  1206
  DCPSInfo  1206
platform independent model (PIM)  1175
platform specific model (PSM)  1175
policy example  1201
publish-subscribe  1176
data local reconstruction layer (DLRL)  1175
data reader
  data-centric publish-subscribe (DCPS)  1179–1180, 1183–1184, 1192–1201, 1203, 1205
data writer
  data-centric publish-subscribe (DCPS)  1178–1179, 1184, 1188–1190, 1195–1200, 1203–1204, 1206
data-centric publish-subscribe (DCPS)  1175
  built-in topic  1179, 1203–1206, 1209
  condition  1180
  data reader  1179–1180, 1183–1184, 1192–1201, 1203, 1205
  data writer  1178–1179, 1184, 1188–1190, 1195–1200, 1203–1204, 1206
  DCPS_DATA_KEY pragma  1184
  DCPS_DATA_TYPE pragma  1184–1185
dcps_ts.pl program  1183–1186, 1207
  DCPSParticipant built-in topic  1179, 1203, 1205
  DCPSPublication built-in topic  1179, 1204
  DCPSSubscription built-in topic  1179, 1205
  DCPSTopic built-in topic  1179, 1203
domain  1178–1179, 1185, 1187, 1190, 1203–1205
domain participant  1178–1179, 1194, 1203–1205
domain participant factory  1187, 1194, 1206
Durability policy  1200
Entity policy  1200
example  1183–1186, 1194–1195, 1205
factory
  domain participant  1187, 1194, 1206
History policy  1199
information repository  1182, 1186, 1206, 1209
  DCPSInfoRepo program  1186, 1194, 1203, 1209
instance  92, 1178, 1182, 1184, 1189–1190, 1196, 1199–1201, 1203–1205
listener  1180, 1183, 1187, 1189, 1192–1193, 1198
Liveliness policy  1197
marshaling  92, 1178–1179, 1182, 1184, 1195
multithreading 1183
overview 1176–1180
pluggable transport 1181–1182, 1188, 1201
policy example 1201
Presentation policy 1200
1199, 1201–1202
quality of service (QoS) 1179–1180, 1183, 1187, 1189, 1196–1201
  supported policies 1197
  unsupported policies 1200
Reliability policy 1198
ResourceLimits policy 1199
sample 1178–1179, 1183–1185, 1189–1190, 1192–1194, 1196, 1198–1202, 1205
simple TCP transport 1181, 1183, 1187–1188, 1191, 1198, 1201–1202
simple UDP transport 1198, 1201–1202
topic 1178–1179, 1183–1184, 1187, 1189–1190, 1192, 1194, 1197–1200, 1203–
1205, 1209
  built-in 1179, 1203–1206, 1209
transport 1181, 1183, 1187, 1191, 1195, 1198, 1201–1202, 1209
  simple TCP 1181, 1183, 1187–1188, 1191, 1198, 1201–1202
  simple UDP 1198, 1201–1202
transport factory 1181, 1188, 1194
wait set 1180

datagram inter-ORB protocol (DIOP) 11, 374, 385–387, 494, 497, 644,
649–650, 725
  DIOP_Factory configuration 386, 725
  endpoints 649
DataReaderListener interface 1192
DCE common inter-ORB protocol (DCE-CIOP) 1075, 1078
DCPS
  See data-centric publish-subscribe (DCPS)
DCPS_DATA_KEY pragma 1184
DCPS_DATA_TYPE pragma 1184–1185
DCPS_TRANS_VERBOSE_DEBUG environment variable 1209
dcps_ts.pl program 1183–1186, 1207
DCPSChunks option 1206–1207
DCPSDebugLevel option 1206
DCPSInfo option 1206
DCPSInfoRepo program 1186, 1194, 1203, 1209
DCPSParticipant built-in topic 1179, 1203, 1205
DCPSPublication built-in topic 1179, 1204
DCPSSubscription built-in topic 1179, 1205
DCPSTopic built-in topic 1179, 1203
DDS_ROOT environment variable 1181, 1183, 1185–1186, 1192, 1207
deadline timeouts 895, 897
debug build flag 1378, 1381
debugging
  ACE_NDEBUG macro 1381
  ORBDebug option 618, 620, 622, 631
  ORBDebugLevel option 620, 622, 632, 1114
  TAO_ORB_DEBUG environment variable 632
decode_profile() operation 481
default proxy factory 337, 339
default resource factory 381, 690, 692, 694–695, 697–698, 705, 711
default.features file 19
default_filter_factory() operation 1001
DefaultValueRefCountBase class 322
delete_contained_entities() operation 1194
DELETE_OBJ macro 488
DELETE_THIS macro 488
demarshaling 96, 133, 157, 572, 586, 627–628
demultiplexing 739
  active 10, 15, 744–745
  dynamic hash 744–745
  linear search 744
  strategies 743
deployment
  COrBA Component Model (CCM) 1213, 1216, 1218, 1224–1226, 1267, 1274–
  1275, 1285–1291, 1312, 1315, 1318, 1323–1325, 1329–1330, 1343–1345,
  1356, 1364, 1370
Deployment And Configuration Engine (DAnCE) 1288, 1290, 1334–
  1335, 1337, 1343, 1345–1346, 1350, 1353–1354, 1364
  reference 1353–1356
deployment descriptor
  COrBA Component Model (CCM) 1217, 1289–1292, 1322
    definition 1288
deployment node
  COrBA Component Model (CCM) 1323
Deshpande, Mayur 1465
design patterns
  See also patterns
Design Patterns: Elements of Reusable Object-Oriented Software
  book 16, 404, 1465
destroy() operation 26, 106–107, 359, 534
destroy_mutex() operation 223
DevGuideExamples directory xxxi
differentiated services 252
codepoints (DSCP) 252
diffserv field (DF) 252
DII
See dynamic invocation interface (DII)
DII COE
CORBA compliance 1446
DIIPollable interface 355–356
DIOP
See datagram inter-ORB protocol (DIOP)
DIOP_Factory configuration 386
direct binding 1138
direct mapping 248–249
directives 594, 602
dynamic 376, 559, 596, 608, 612
components 596
resume 603, 614
static 559, 595, 608, 690, 739, 763, 847, 915, 941
suspend 603, 614
DiscardPolicy property 961, 963–964, 966–967, 1026
disconnect_push_consumer() operation 831, 837, 843, 878, 901
disconnect_push_supplier() operation 829–830, 837, 876, 901
disconnect_sequence_push_consumer() operation 1011
disconnect_sequence_push_supplier() operation 1006
disconnect_structured_push_consumer() operation 986–988
disconnect_structured_push_supplier() operation 979
disjunction group 889–890
dispatching
events 847
module 873
real-time event service (RTES) 873
options
CECDDispatching 847–848, 851, 856–857
CECDDispatchingThreads 847–848, 856–857
ECDispatching 915–916, 920, 924–926
ECDispatchingThreads 916, 924, 926
strategies
event channel 847
multithreaded 847, 916
priority 916
reactive 847, 916
distributable thread 213, 217, 235–238, 240
distribution model
    pull 826
    push 826
DLRL
See data local reconstruction layer (DLRL)
DNS
See domain name servers (DNS)
do() operation 237, 273
DOC Group xxvi, xxxii–xxxiii, 3, 1065–1066, 1445, 1447
domain
data-centric publish-subscribe (DCPS) 1178–1179, 1185, 1187, 1190, 1203–1205, 1209
domain name servers (DNS) 377, 634, 642, 655–656
domain participant
data-centric publish-subscribe (DCPS) 1178–1179, 1194, 1203–1205
domain participant factory
data-centric publish-subscribe (DCPS) 1187, 1194, 1206
DSI
See dynamic skeleton interface (DSI)
duplicate() operation 475
dynamic hash 745, 751, 754, 756–757, 760, 762
dynamic invocation interface (DII) 10, 12, 140, 354
dynamic scheduling 14, 211, 213, 235, 260
dynamic service example 612
dynamic skeleton interface (DSI) 10, 12
dynamic threads 224
DynamicAny module 355

eager resource usage strategy 691, 706–707
earliest deadline first (EDF) scheduling 243
EC Factory configuration 915–916
ECDDispatching option 915–916, 920, 924–926
ECDDispatchingThreads option 916, 924, 926
ECFiltering option 887, 892, 915, 917, 927
ECObserver option 901, 917, 928
ECProxyConsumerLock option 918, 929
ECProxyPushConsumerCollection option 919, 930
ECProxyPushSupplierCollection option 919, 932
ECProxySupplierLock option 918, 934
ECScheduling option 917, 927, 935
ECSupplierFilter option 917
ECSupplierFiltering option 917, 938
ECTimeout option 916, 939
ECUseORBid option 917, 940
EDF_Scheduling module 243
em3 build type 36, 69
emits keyword
  CORBA Component Model (CCM) 1224, 1227, 1299
enable_network_priority attribute 231, 252
encode_endpoints() operation 486
end_scheduling_segment() operation 238, 273
endpoint() operation 485
endpoint_count() operation 486
endpoint-specific options 650
end-to-end priority propagation 14, 213, 227
entity composition
  CORBA Component Model (CCM) 1229
entity keyword
  CORBA Component Model (CCM) 1231
environment specific inter-ORB protocol (ESIOP) 371–372
DCE 371
environment variables 22, 1392
ACE_ROOT 22, 84, 88, 97, 1333, 1349, 1352, 1365, 1392
BOOST_INCLUDE 1352
BOOST_LIB 1352
BOOST_ROOT 1352
CIAO_ROOT 1290, 1331–1332, 1345, 1349–1350, 1352–1353, 1357, 1365
DCPS_TRANS_VERBOSE_DEBUG 1209
DDS_ROOT 1181, 1183, 1185–1186, 1192, 1192, 1207
ImplRepoServiceIOR 1140, 1144, 1147, 1156, 1165, 1167
ImplRepoServicePort 653, 788, 1169
InterfaceRepoServicePort 788, 1052
InterfaceRepositoryIOR 839, 1052, 1054, 1063
LD_LIBRARY_PATH 22, 597, 1294, 1331
MAKEFLAGS 112, 885
MPC_COMMANDLINE 42
MPC_DEPENDENCY_COMBINED_STATIC_LIBRARY 42
MPC_LOGGING 42
MPC_SILENT 42
MPC_VERBOSE_ORDERING 42
NameServiceIOR 800, 828, 874
NameServicePort 665, 788, 799
on UNIX 22
on Windows 22
OPENSSL_CONF 1099–1100, 1106
PATH 597, 605, 634, 648–649, 655–656, 671, 682, 752, 1106, 1294, 1332, 1349,
1352, 1382
SSL_CERT_DIR 1111–1112
SSL_CERT_FILE 1111–1112
SSL_EGD_FILE 1111
SSL_RAND_FILE 1111
SSL_ROOT 1107–1108
TAO_IDL_PREPROCESSOR 86, 88
See also IDL compiler
TAO_IDL_PREPROCESSOR_ARGS 88
See also IDL compiler
TAO_ORB_DEBUG 632
TAO_ORBENDPOINT 620, 642, 646
TAO_ORBSVCS 1398
TAO_ROOT 22, 84, 86, 88, 1392, 1406, 1413
TAO_USE_IMR 1138–1139, 1144, 1147
TradingServicePort 684, 788
UTILITY_ROOT 1349, 1352
XERCESCROOT 1331, 1346–1347, 1349, 1351–1352

error handling 101–137
ACE exception handling macros 112
ACE_TRY/ACE_CATCH blocks
  multiple 124
  nested 126
errno 136
error number codes 136
guidelines 127
simulating exceptions 122
TAO error number codes 136
throwing exceptions with ACE_THROW 119
without exception handling 110

ESIOP
See environment specific inter-ORB protocol (ESIOP)

establish_components() operation 306
ETCL
See extended trader constraint language (ETCL)

etherealize 629

event
CORBA Component Model (CCM) 1214–1215, 1217, 1219–1220, 1222–1227, 1244, 1248, 1252, 1254, 1266–1268, 1270, 1307

event channel 875
attributes 900
  consumer_poa 836
  consumer_reconnect 836–837
  disconnect_callbacks 836–837
  supplier_poa 836
  supplier_reconnect 836–837
collocating 834
consumer and supplier control options 848
consumer proxy 829
creation and destruction 827, 834
disconnecting consumer 837, 853
disconnecting supplier 829, 837, 866
example 898
implementation 827
locating 834
locking options 848
managing own servants 834
object 828

options
  CECConsumerControl 849, 853–855, 861, 868
  CECConsumerControlPeriod 849, 854–855, 861, 867–868
  CECConsumerControlTimeout 849–850
  CECDispatching 847–848, 851, 856–857
  CECDispatchingThreads 847–848, 856–857
  CECProxyConsumerCollection 850, 858–859
  CECProxyConsumerLock 848, 860
  CECProxyDisconnectRetries 849
  CECProxySupplierCollection 850, 862–863
  CECProxySupplierLock 848, 864
  CECReactivePullingPeriod 852, 865
  CECSupplierControl 849, 866–867
  CECSupplierControlPeriod 849, 867
  CECSupplierControlTimeout 849–850
  CECUseORBId 852, 869
  ECDisplaying 915–916, 920, 924–926
E

ECDispatchingThreads 916, 924, 926
ECFiltering 887, 892, 915, 917, 927
ECObserver 901, 917, 928
ECProxyConsumerLock 918, 929
ECProxyPushConsumerCollection 919, 930
ECProxyPushSupplierCollection 919, 932
ECProxySupplierLock 918, 934
ECScheduling 917, 927, 935
ECSupplierFilter 917
ECSupplierFiltering 917, 938
ECTimeout 916, 939
ECUseORBId 917, 940
proxy 828
resource factory 846, 915
EC_Factory 915
options 852, 921
setting attributes 836, 900
supplier proxy 833
typed 827
creating 839, 846
destroying 846
example 838
untyped 827
event channel attributes 900
event consumer
CORBA Component Model (CCM) 1227, 1293, 1297, 1299, 1306–1307, 1340
event correlation
real-time event service (RTES) 873
event emitter
CORBA Component Model (CCM) 1224, 1297, 1299
event filtering 825, 954
adding to consumer 1003
adding to supplier 1001
notification service 954
event publisher
CORBA Component Model (CCM) 1293, 1297, 1299–1300
event service xxx, xxxvi, 13, 825–869, 872, 876, 952, 959
and naming service 845–846
compatability with notification service 1021
connecting to the channel 832
CosEvent_Service program 827, 834
creating an event channel 827
creating servants 834
decoupled supplier consumer 825
dispatching 847
event filtering 825
Event_Service program 875
example 827, 834
ORB service libraries 784
overview 826
proxy collection options 850, 919
pull model support 838
push
  consumer interface 831
distribution model 826
  supplier interface 830
pushing events 828
red-black tree 850
starting CosEvent_Service server 828
structured event types 825
supplier 825
typed
  and interface repository 839
typed event channel 827, 838
  creating 839, 846
  destroying 846
untyped event channel 827

event sink
  CORBA Component Model (CCM) 1216

event source
  CORBA Component Model (CCM) 1216

event structure
  notification service 951–952
  real-time event service 883
  real-time event service (RTES) 883

event supplier
  notification service 980

event type
  CORBA Component Model (CCM) 1215, 1219, 1221–1222, 1224–1225, 1227,
  1244, 1300, 1307, 1335, 1360

event type factory
  CORBA Component Model (CCM) 1222, 1360

event_handler_i() operation 510
**Event_Service program** 875, 913  
command line options 913  

**EventBase interface** 1222  

**EventBatch data type** 954  

**EventChannel interface** 876, 899, 902, 906, 914, 950, 956, 963, 983, 989, 1004, 1008, 1013, 1016  

**EventHeader structure**  
members 883  
real-time event service (RTES) 883  

**EventReliability property** 962, 967, 972, 976, 1045  

**EventSourceID** 876, 883  

**EventType** 877, 883–884  

**eventtype keyword**  
CORBA Component Model (CCM) 1222, 1227  
value types 318–319, 325, 330  

**example** xxix, 22, 160–167, 287  
asynchronous method handling (AMH) 188, 191  
asynchronous method invocation (AMI) 160  
building 30  
collocated notification channel 1015  
Component-Integrated ACE ORB (CIAO) 1219  
CORBA Component Model (CCM) 1219  
corbaloc object URL 802  
corbaname object URL 804  
data-centric publish-subscribe (DCPS) 1183–1186, 1194–1195, 1205  
policies 1201  
DevGuideExamples directory xxxi  
event channel  
local 898  
event service collocation 834  
exceptions 116, 128–129  
getting started 21  
IDL compiler 85, 1358  
implementation repository  
activator 1148  
basic indirection 1141  
ImplRepo_Service program 1170  
ImR_Activator program 1173  
IOR table 1153  
tao_imr utility 1165  
interface repository 1055  
IORTable 366
local interface 360
local interfaces 360
locality constrained 360
logging 344
MakeProjectCreator (MPC) 63, 74
messaging 160
Messenger interface 23
Messenger_i implementation class 24
MessengerClient 23, 26
MessengerServer 23, 25
multiple protocol endpoint 651
multithreading 557, 562, 565
naming service persistence 823
naming service 794
notification service 977
pluggable protocols 405, 411
portable interceptors 307
  client-side recursion 312
  simple authentication interceptor 287
  using the codec 300
real-time CORBA 218, 270, 278
real-time event service (RTES) 874
  local event channel 898
running 30
security 1115
send_message() operation 26
servant manager 356
ServantLocator class 356
service configurator 613
setting up your environment 22
smart proxies 345–351
tao_idl2_to_idl3 compiler 1366
TAO_Naming classes 811, 823
typed event channel 838–839
value type 320

ExceptionHolder class 142, 145–146, 148

alternatives 110
BAD_INV_ORDER 135, 541–542, 545
BAD_PARAM 134, 309
BAD_QOS 969
catching 104, 107
COMM_FAILURE 103, 107, 117, 133
completed() operation 105
CORBA 103
example 116, 128–129
INCOMPATIBLE_SCHEDULING_DISCIPLINES 242
INV_OBJREF 103, 133
location code 132
MARCHAL 316, 354, 626
minor codes 131
minor() operation 105, 131
multiple blocks 124
MutexNotFound 259
nested blocks 126
NO_IMPLEMENT 355
OBJECT_NOT_EXIST 103, 358, 542
real-time CORBA
INCOMPATIBLE_SCHEDULING_DISCIPLINES 242
MutexNotFound 259
returning values 122
side effects 118, 129
simulating 112
system 103
throwing 119
TIMEOUT 133–134
TRANSIENT 103, 107, 132–134, 672, 745–747, 756, 760, 1117
UNKNOWN 133
user 108
WrongPolicy 272

exceptions build flag 1377, 1379
exclude option 37
ExecutionManager program 1343, 1354–1356
executor
CORBA Component Model (CCM) 1215, 1229, 1231, 1233–1239, 1241–1242,
1244–1245, 1248–1251, 1253–1269, 1271, 1273, 1275, 1277–1279, 1285,
1292, 1296, 1304, 1306, 1308, 1311, 1330, 1336–1337, 1339–1340, 1344–
1345, 1356–1357, 1359, 1362–1363
executor keyword
CORBA Component Model (CCM) 1231
expand_var option 37
explicit binding 10, 213, 228
extended trader constraint language (ETCL) 955, 1001, 1003
facet

factory  835
   advanced resource  380, 710
   initialization  741
   server strategy  134, 556, 558, 560–562, 593, 595–596, 620, 739–742, 744, 746–762
   value type  319–321, 323–324, 327

factory keyword
   value types  320

fast build flag  1378, 1381

feature packages
   common secure interoperability (CSI)  1066

feature_file option  37, 60

features option  37

federating event channels
   IP multicast  911
   mechanism  913
   real-time event service (RTES)  902
   selection mechanism  913
   UDP  907

FIFO
   See first in first out (FIFO) strategy

filtering
   adding to consumer  1003
   adding to supplier  1001
   by event type  887
   by source ID  888
   construction  894
   correlation  886
   disjunction groups  889
   example  890
   InterFilterGroupOperator parameter  956, 983–984, 990
   notification service  954, 1001, 1003
real-time event service 888

**find_topic() operation** 1190

**fini() operation** 493, 608, 610

**first in first out (FIFO) strategy** 691, 696

**fixed priority scheduling** 235, 242

**fl_reactor build flag** 60, 711, 1378, 1384

**flushing strategies** 571–572, 575–576, 695
  - blocking 576
  - leader-follower 575
  - reactive 575

**footprint** 594, 627, 710, 754, 938, 1391, 1397
  - minimizing library size 1397
  - reduction 5, 110, 156, 354, 580, 594, 627, 710, 754, 772, 938
    - minimum_corba build flag 182–183, 262, 282, 1378, 1383, 1385–1386

**forward declaration** 1051

**FP_Scheduling module** 243

**FP_Scheduling::FP_Scheduler interface** 243

**FP_Scheduling::SegmentSchedulingParameterPolicy interface** 243

**Free Software Foundation**
  GNU Public License (GPL) xxxix, xli

**frequently asked questions (FAQ)** xxxii–xxxiii

---

**G**

**Gamma, Erich** 16, 404, 1465

**Gang of Four (GoF)**
  See *Design Patterns: Elements of Reusable Object-Oriented Software* book

  - bi-directional 12, 14, 139–140, 180–183, 199
    - BidirectionalPolicyValue 181
    - BiDirPolicy 183
    - security 183
  - lite 373
  - version 1.2 181

**generate_request_header() operation** 512

**generic security service (GSS) protocol** 1079

**genins option** 37

**get_client_policy() operation** 355

**get_connections_content() operation** 1283
get_domain_managers() operation 355
get_effective_component() operation 307, 309
get_event_channel() operation 983, 989
get_interface() operation 355
get_parser_names() operation 700
get_peer_certificate() operation 1125
get_peer_certificate_chain() operation 1125
get_policy() operation 355, 1077
get_policy_overrides() operation 355
get_protocol_factories() operation 699
get_proxy() operation 337–338
get_qos() operation 964, 968
get_slot() operation 311
getting started 21–31
ghs build type 36
GIOP
   See general inter-ORB protocol (GIOP)
global option 37
GNU Make xxxi, 18–19, 27–30, 36, 112, 262, 1377, 1379–1388, 1396, 1418, 1426–1427, 1430
   build flags
      ami xxx, 1378, 1385
corba_messaging 262, 1378, 1385
debug 1378, 1381
exceptions 112–113, 1377, 1379
fast 1378, 1381
fl_reactor 60, 711, 1378, 1384
gtk_reactor 1378, 1384
include_env 112, 1377, 1380
inline 49–50, 52, 54–55, 62, 65, 85, 89, 1360, 1378, 1380
interceptors xxx, 5, 354, 629, 1074, 1077, 1378, 1385
minimum_corba 1378, 1383, 1385
optimize 1378, 1381
pipes 1379, 1386
probe 1378, 1383
profile 1378, 1383
purify 1378, 1382
qt_reactor 1378, 1384
quantify 1071, 1378, 1382
repo 1378, 1381
rt_corba 262, 1378, 1385
rtti 1377, 1380
H

rwho 1378, 1386
sctp 1379, 1387
shared_libs 1378, 1382–1383
shared_libs_only 1378, 1383
smart_proxies 335, 341
split 1379, 1387
ssl 1107, 1378, 1386
static_libs 1378, 1382–1383
static_libs_only 1378, 1383
static_link 1379, 1388
stlport 1378–1379, 1386
templates 1388
threads 1378, 1381
tk_reactor 712, 737, 1378, 1384
versioned_so 1379, 1387
wfmo 1379, 1387
winregistry 1379, 1388
xt_reactor 1378, 1383
gprof program 1383

GNU Public License (GPL) xxxix, xli
gnuace build type 28, 36, 46, 48, 50, 69

GoF
See Design Patterns: Elements of Reusable Object-Oriented Software book

Gokhale, Aniruddha 1467
gperf program 1369
gprof program 1383
gtk_reactor build flag 1378, 1384

H

half-sync/half-async pattern 15
handle_close() operation 467
handle_events() operation 518, 569–570, 581
handle_input() operation 410–411, 414, 423–424, 467, 471, 508, 518, 524, 570
handle_output() operation 470
handle_timeout() operation 471, 545
Harris, Timothy 1465
hash() operation 355, 476, 482
hashing
   perfect 10, 15
**Helm, Richard** 16, 404, 1465  
**Henning, Michi** xxvii, xxxii, 1465  
**Herron, Thomas** 372, 1467  
**hierarchy option** 38  
**home executor**  
CORBA Component Model (CCM) 1229–1231, 1233, 1260–1264, 1267, 1272, 1283, 1292  
**home keyword**  
CORBA Component Model (CCM) 1224, 1227  
**hostname_in_ior option** 379, 396, 398, 645  
**HTBP**  
*See HTTP tunneling bi-directional protocol (HTBP)*  
**HTIOP**  
*See HTTP tunneling inter-ORB protocol (HTIOP)*  
**HTTP**  
*See hypertext transfer protocol (HTTP)*  
**HTTP tunneling bi-directional protocol (HTBP)** 393  
**HTTP tunneling inter-ORB protocol (HTIOP)** xxviii, 11, 374, 393  
**Huston, Stephen D.** 16, 404, 525, 534–535, 1466, 1468  
**hypertext transfer protocol (HTTP)** 11  

---

**id_to_reference() operation** 26, 409  
**IDL**  
*See interface definition language (IDL)*  
**IDL compiler** 9, 23, 83–100, 102, 142  
A option 87  
Ce option 100  
ci option 89  
cs option 89  
Cw option 99  
D option 87  
d option 99  
E option 87  
exception handling 93  
GA option 92  
GC option 92  
Gd option 96  
Gdcps option 92, 1182, 1184, 1186
<table>
<thead>
<tr>
<th>Option</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ge</td>
<td>93</td>
</tr>
<tr>
<td>GH</td>
<td>92, 190</td>
</tr>
<tr>
<td>GI</td>
<td>23, 90, 345, 361</td>
</tr>
<tr>
<td>Gla</td>
<td>90</td>
</tr>
<tr>
<td>Glb</td>
<td>90</td>
</tr>
<tr>
<td>Glc</td>
<td>90</td>
</tr>
<tr>
<td>Gle</td>
<td>90</td>
</tr>
<tr>
<td>Glh</td>
<td>90</td>
</tr>
<tr>
<td>GIs</td>
<td>90</td>
</tr>
<tr>
<td>Gp</td>
<td>96</td>
</tr>
<tr>
<td>Gsp</td>
<td>92, 335, 337, 339–341, 343, 345</td>
</tr>
<tr>
<td>GT</td>
<td>92</td>
</tr>
<tr>
<td>Gt</td>
<td>92</td>
</tr>
<tr>
<td>Guc</td>
<td>92</td>
</tr>
<tr>
<td>H</td>
<td>95</td>
</tr>
<tr>
<td>hc</td>
<td>89</td>
</tr>
<tr>
<td>hI</td>
<td>89</td>
</tr>
<tr>
<td>hs</td>
<td>89</td>
</tr>
<tr>
<td>hT</td>
<td>89</td>
</tr>
<tr>
<td>I</td>
<td>87</td>
</tr>
<tr>
<td>ic</td>
<td>92</td>
</tr>
<tr>
<td>in</td>
<td>92</td>
</tr>
<tr>
<td>o</td>
<td>89</td>
</tr>
<tr>
<td>operation lookup strategy</td>
<td>95</td>
</tr>
<tr>
<td>options</td>
<td>86–100</td>
</tr>
<tr>
<td>back-end processing</td>
<td>96</td>
</tr>
<tr>
<td>export_include</td>
<td>97</td>
</tr>
<tr>
<td>export_macro</td>
<td>97</td>
</tr>
<tr>
<td>obv_opt_accessor</td>
<td>98</td>
</tr>
<tr>
<td>pch_include</td>
<td>97</td>
</tr>
<tr>
<td>post_include</td>
<td>98</td>
</tr>
<tr>
<td>pre_include</td>
<td>97</td>
</tr>
<tr>
<td>skel_export_include</td>
<td>97</td>
</tr>
<tr>
<td>skel_export_macro</td>
<td>97</td>
</tr>
<tr>
<td>stub_export_include</td>
<td>97</td>
</tr>
<tr>
<td>stub_export_macro</td>
<td>97</td>
</tr>
<tr>
<td>code generation</td>
<td>92</td>
</tr>
<tr>
<td>code suppression</td>
<td>98</td>
</tr>
<tr>
<td>collocation strategy</td>
<td>96</td>
</tr>
<tr>
<td>output and reporting</td>
<td>99</td>
</tr>
<tr>
<td>output files</td>
<td>84–85, 89</td>
</tr>
<tr>
<td>pragma ident</td>
<td>88</td>
</tr>
</tbody>
</table>
pragma prefix  88  
pragma version  88  
preprocessing  86, 1368  
Sa option  98  
Sc option  98  
Sd option  98  
sl option  89  
si option  89  
Sm option  99  
smart proxies  332, 340  
Sp option  98  
ss option  89  
St option  98  
st option  89  
starter code  90  
t option  99  
U option  87  
V option  99  
v option  99  
w option  99  
Wb option  96  
Wp option  87  
Yp option  87  

**IDL2**

*See interface definition language (IDL), IDL2*

**IDL3**

*See interface definition language (IDL), IDL3*

**IDL3-to-IDL2 compiler**  1364–1365, 1367–1369

building  1365  
Ce option  1369  
Cw option  1369  
d option  1369  
g option  1369  
input files  1366  
o option  1369  
output files  1366  
preprocessing  1368  
reference  1364–1369  
t option  1369  
V option  1369  
v option  1369
w option 1369

IETF
See Internet Engineering Task Force (IETF)

IFR_Service program 1051–1052, 1054, 1062–1063
multithreaded 1052

IIOP 434
See internet inter-ORB protocol (IIOP)

IIOP Complete: Understanding CORBA and Middleware Interoperability book 1467

implementation artifact descriptor 1289, 1292–1297, 1302, 1304, 1306, 1308–1309, 1311

implementation repository 366, 624, 653, 1167, 1171
and IOR table 1151

directives 1160

directives
activator 1148
basic indirection 1141
ImplRepo_Service program 1170
ImR_Activator program 1173
IOR table 1153
tao_imr utility 1165
ImplRepo_Service program 636, 653, 685
ImplRepoServiceIOR environment variable 1140, 1144, 1147, 1156, 1165, 1167
ImplRepoServicePort 653, 788, 1169
ImR_Activator program 685, 1148

options
ORBImpRepoServicePort 624, 653, 788
server start-up 1145
tao_imr utility 1160
activate command 1161
add/update commands 1162
autostart command 1163
ior command 1163
list command 1164
remove command 1164
shutdown command 1164
shutdown-repo command 1165
TAO_USE_IMR environment variable 1138–1139, 1144, 1147

implies keyword
CORBA Component Model (CCM) 1231

ImplRepoServiceIOR environment variable 1140, 1144, 1147, 1156, 1165, 1167
ImplRepoServicePort environment variable 653, 788, 1169
include option 38
include_env build flag 112, 1377, 1380
INCOMPATIBLE_SCHEDULING_DISCIPLINES exception 242
indirect binding 366, 1138, 1151
info() operation 614, 616
information repository
data-centric publish-subscribe (DCPS) 1182, 1186, 1206, 1209
init() operation 493–494, 608, 810–811
init_protocols_factories() operation 699
inline build flag 49–50, 52, 54–55, 62, 65, 85, 89, 1360, 1378, 1380
input_cdr_allocator_type_locked() operation 694
input_cdr_buffer_allocator() operation 693
input_cdr_dblock_allocator() operation 693
input_cdr_msgblock_allocator() operation 693
installation
ACE and TAO 1389
system requirements 18
See also building, ACE and TAO
instance
data-centric publish-subscribe (DCPS) 92, 1178, 1182, 1184, 1189–1190, 1196, 1199–1201, 1203–1205
instance() operation 338, 599, 615
Institute for Software Integrated Systems (ISIS) 3
interceptors
See portable interceptors
interceptors build flag 282, 1378, 1385
interface definition language (IDL) xxvi, xxxv, xli–xlii, 9, 23, 150
back-end options 96
C++ mapping 5, 83, 102
collocation strategy options 96
compiler
See IDL compiler
compiler front end (CFE) xli–xlii
definitions 160
environment variables 88
gperf program 1369
IDL2 1215, 1221
IDL3 1215, 1222–1223
preprocessing options 86, 1368
pseudo (PIDL) 173, 178, 217, 282, 284–285, 305, 311, 1386
starter implementation files 90
stubs and skeletons  9
suppression options  98
tao_idl program  23–24, 53, 83–91, 95, 97–98, 162, 1184, 1368
tao_idl3_to_idl2 program  1365–1366, 1368–1369
interface repository  xxx, 13, 104, 1049–1051, 1055–1056, 1062–1063, 1397
command line options  1051
compliance  1437, 1440
example  1055
forward declaration  1051
IFR_Service program  1051–1052, 1054, 1062–1063
implementation  1051
InterfaceDef interface  1050
InterfaceRepoServicePort  788, 1052
Repository object  1050
RepositoryId  285–286, 1050–1051
tao_ifr program  1051, 1053–1054, 1062–1063
options  1053–1054
InterfaceRepoServicePort environment variable  788, 1052
InterfaceRepositoryIOR environment variable  839, 1052, 1054, 1063
interfaces
Components::EventBase  1222
CORBA::Current  157, 221, 268, 311, 355, 1125
CORBA::DIIPollable  355–356
CORBA::DynamicAny  355
CORBA::InterfaceDef  1050
CORBA::LocalObject  357, 360, 363, 1239
CORBA::LocalObject class  306
CORBA::Policy  356
CORBA::PolicyCurrent  157, 170, 355–356, 654
CORBA::PolicyList  169, 171–172, 174, 176, 179, 182, 219, 224, 226, 229–230,
CORBA::PolicyManager  170, 172, 176, 179, 254, 355–356
CORBA::Pollable  355–356
CORBA::PollableSet  355–356
CosEventComm::PushConsumer  831
831
CosEventComm::PushSupplier  830
CosNaming::NamingContextExt  807–808
CosNotifyChannelAdmin::ConsumerAdmin  950, 958, 989–990, 996, 1003, 1013
CosNotifyChannelAdmin::EventChannel  956, 963, 983, 989, 1004, 1008, 1013,
1016
CosNotifyChannelAdmin::ObtainInfoMode  999–1000
CosNotifyChannelAdmin::ProxyConsumer 955, 984, 999, 1009
CosNotifyChannelAdmin::ProxySupplier 955, 990, 999, 1013–1014
CosNotifyChannelAdmin::SequenceProxyPushConsumer 951, 954, 982, 984, 986, 1005, 1009, 1011, 1014
CosNotifyChannelAdmin::SequenceProxyPushSupplier 951, 979, 985, 990, 1005, 1013–1014
DataReader
type-specific 1185, 1193
DataReaderListener 1192
DataWriter
type-specific 1185
FP_Scheduling::FP_Scheduler 243
FP_Scheduling::SegmentSchedulingParameterPolicy 243
IORTable::Locator 368
IORTable::Table 366
ORBInitializer 286, 297, 307, 313
ORBInitInfo 286
PortableInterceptor::IORInfo 306, 308
PortableInterceptor::PICurrent 284, 311–314, 316
PortableServer::ServantLocator 356–357, 359
RTCORBA::Current 237
RTCORBA::Mutex 240
RTCORBA::NetworkPriorityMapping 256–258
RTCORBA::RTORB 215
RtecEventChannelAdmin::EventChannel 876, 899, 902, 906, 914
RtecEventComm::PushConsumer 878, 880
RtecEventComm::PushSupplier 876
RTScheduling::Current 217, 235–238, 260–261, 273
RTScheduling::DistributableThread 235
RTScheduling::ResourceManager 240, 242
RTScheduling::Scheduler 240–241, 260
RTScheduling::ThreadAction 236, 273, 275
TypeSupport
type-specific 1185
InterFilterGroupOperator parameter 956, 983–984, 990
Internet Engineering Task Force (IETF) 252, 396
endpoints 646
example 650
IIOP_Factory configuration 376, 725
IIOP_Lite_Factory configuration 725
interoperable object reference (IOR) 642, 671, 681, 700, 749–750, 752, 764, 794, 900
custom parsers 699
formats 657, 700
interceptors 305, 316
profile locking 764
inter-process communications (IPC) xxxvi, 231, 371, 374, 380, 382–383, 525, 648, 671, 725–726
local 11, 644
interval timeouts
real-time event service (RTES) 895–896
Introduction to CORBA course xxxv
Introduction to Model-Driven Architecture (MDA) course xxxvii
Introduction to Real-Time Systems course xxxvi
invocation
asynchronous 234
dynamic 12
oneway 14, 140–141, 175–179, 221, 234, 332, 386–387, 389, 392, 531, 544, 579, 649
reliable oneway 234
static 9, 12
invoke() operation 511
IOR
See interoperable object reference (IOR)
IOR table 366
IORInfo interface 306, 308
IORTable
example 366
IORTable::Locator interface 368
IORTable::Table interface 366
is_a() operation 355, 367
is_equivalent() operation 355, 474
is_nil() operation 27, 541
ISIS
See Institute for Software Integrated Systems (ISIS)
JacORB xxxiv–xxxv
Java Reflection API 1049
jitter 4
Johnson, James CE 1466
Johnson, Ralph 16, 404, 1465

K

Kachroo, Vishall 1467
Kerberos 1075, 1079, 1089
kernel and system configuration 1423
   VxWorks 1415
Klinker, Paul 372, 1467
Kuhns, Fred 1466

L

language option 38
latency 4
lazy resource usage strategy 691, 706–707
LD_LIBRARY_PATH environment variable 22, 597, 1294, 1331
leader/followers
   pattern 563
   strategy 576, 695, 719
least frequently used (LFU) strategy 691, 696, 712, 733
least laxity first (LLF) scheduling 244
least recently used (LRU) strategy 691, 696–697, 712, 733
Levine, David 1465
Lewis, Bil 1466
LFU
   See least frequently used (LFU) strategy
libraries
   minimizing size 1397
licensing xxv–xxvi, xxxix, xli–xlvi
   Free Software Foundation
      GNU Public License (GPL) xxxix, xli
life cycle service 13, 784
    ORB service libraries 784
Linux xxxiv, 21, 507, 1107
Linux kernel stream control transmission protocol (lksctp) 397
listener
data-centric publish-subscribe (DCPS) 1180, 1183, 1187, 1189, 1192–1193, 1198
lite protocols 726
LLF_Scheduling module 244
load balancing service 13, 334, 642, 764, 784
    ORB service libraries 784
local interfaces 318, 353–355, 357, 360, 363
    _add_ref() operation 363
    _remove_ref() operation 363
    asynchronous method invocation (AMI) 354
    C++ mapping 354
    CORBA Component Model (CCM) 353
    CORBA::LocalObject 354–355, 357
    dynamic invocation interface 354
    example 360
    object 353
    ORB mediation 354
    postinvoke() operation 356
    preinvoke() operation 356
    reference counting 363
    servant activation 356
    servant locator 356
locality constrained 354–355, 357, 360, 363
    example 360
LocalObject interface 369, 1239
locate() operation 368
location codes 132
LOCATION_FORWARD 132, 283, 368, 764, 771
lock() operation 240
locked_transport_cache() operation 696–697
locking
    lock() operation 222
    option
        real-time event service (RTES) 918
    strategies 591
log service 13
    ORB service libraries 784
logical AND groups
real-time event service (RTES) 891

LRU
See least recently used (LRU) strategy

LynuxWorks, Inc. 1423
LynxOS iv, 776, 1390, 1423, 1425, 1429–1430, 1432, 1447
cross compilation 1425
kernel and system configuration 1423
using ACE and TAO 1423

M

magic number 379, 403, 725
MAIN_THREAD_MODEL policy 547, 550–551, 553–554
make build type 36
make_acceptor() operation 494
make_coexistence option 38
make_connector() operation 495
make_mprofile() operation 483
make_svc_handler() operation 423
MAKEFLAGS environment variable 112, 885
MakeProjectCreator (MPC) xliii, 19, 21, 37, 41, 76, 1392, 1395
adding a new type 65
apply_project option 38
base option 37
base project files 43
base workspace files 43
build type
automake 36
bmake 36
cbx 36
e3 36, 69
ghs 36
gnuace 28, 36, 46, 48, 50, 69
make 36
nmake 19, 27, 36, 38, 44, 51
sle 36
vc6 29, 36, 41, 44, 51, 62, 69
vc7 36, 44, 51, 69
vc71 29, 36, 51
vc8 36
command line 37
complete option 39
custom types 70
defaults 61
documentation 63
eexample 63, 74
exclude option 37
expand_var option 37
feature file 59
feature_file option 37, 60
features option 37
genins option 37
global option 37
header files 62
hierarchy option 38
IDL files 63
include option 38
inline files 62
input file syntax 43
language option 38
Make Project Creator (mpc) 21, 27–28, 31
Make Workspace Creator (mwc) 28–29
make_coexistence option 38
mpc.pl program 34–35, 37–38, 41, 79
mwc.pl program 28–29, 35–38, 40–41, 44, 79
name_modifier option 38
noreldefs option 38
notoplevel option 38
options
apply_project 38
base 37
complete 39
exclude 37
expand_var 37
feature_file 37, 60
features 37
genins 37
global 37
hierarchy 38
idlflags 86
include 38
language 38
make_coexistence 38
name_modifier 38
noreldefs 38, 44
notoplevel 38
recurse 38, 44
relative 39
static 39
template 38–41, 49–50, 52, 54, 62, 64–66, 68, 72, 74, 77
ti 39–41
type 39
use_env 39
value_project 39, 41
value_template 39, 41
version 39

ORB services base projects 61
recurse option 38
relative option 39
resource files 63
rt_client base project 262
rt_server base project 262
source files 61
static option 39
supported build tools 36
template 74
template option 39
ti option 39
type option 39
use_env option 39
value_project option 39
value_template option 39
version option 39

manages keyword
  CORBA Component Model (CCM) 1224, 1227, 1231

managing connections
  real-time event service (RTES) 880

mapping
  C++ language mapping 5
direct 248–249

mapping() operation 251
MARSHAL exception 316, 354, 626
marshaling 9, 96, 133, 157, 300, 372–373, 402, 572, 586, 622, 626–628, 693, 885
data-centric publish-subscribe (DCPS) 92, 1178–1179, 1182, 1184, 1195

**Martin, Robert C.** 1466

**match_prefix() operation** 495–496, 700

**max_muxed_connections() operation** 696–697

**Max_Utility_Scheduling module** 245

**maximize accrued utility (MAU) scheduling** 245

**maxPriority constant** 220

**mcast option** 635

**messaging** 139–183, 185, 372

**_validate_connection() operation** 174

**activation-per-AMI-call strategy** 156–157

**associating replies with requests** 156

**asynchronous method invocation (AMI)** 1467

**asynchronous requests** 177

**bi-directional general inter-ORB protocol (GIOP)** 180

**BidirectionalPolicyValue** 181

**BUFFER_MESSAGE_COUNT** 178–179

**BUFFER_TIMEOUT** 178

**buffered oneway request** 177

**BufferingConstraint policy** 177–179

**callback solution** 141

**client-side policy management** 170

**connection timeout** 173–174

**ConnectionTimeout policy** 174

**controlling the delivery of AMI-based requests** 167

**corba_messaging build flag** 262

**create_policy() operation** 168–169, 172, 174, 176, 179, 182

**creating a reply-handler class** 150

**drawbacks to using AMI** 142

**exceptions**

**ExceptionHolder class** 142, 145–146, 148

**replies** 148

**reply-handler functions** 155

**introduction** 1467

**oneway requests** 177

**policies**

**creating** 168

**destroying** 171

**management of** 168

**quality of service (QoS)** 171

**reply**

**handler** 147, 150


<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeouts</td>
<td>171</td>
</tr>
<tr>
<td>request</td>
<td></td>
</tr>
<tr>
<td>delivery</td>
<td>167</td>
</tr>
<tr>
<td>timeouts</td>
<td>171</td>
</tr>
<tr>
<td>secure</td>
<td>1073</td>
</tr>
<tr>
<td>sendc_prefix</td>
<td>141, 143</td>
</tr>
<tr>
<td>server-differentiated-reply strategy</td>
<td>156–157</td>
</tr>
<tr>
<td>server-side policy management</td>
<td>171</td>
</tr>
<tr>
<td>specification</td>
<td>211</td>
</tr>
<tr>
<td>SYNC_NONE</td>
<td>179</td>
</tr>
<tr>
<td>synchronous IDL operation</td>
<td>146</td>
</tr>
<tr>
<td>SyncScope policy</td>
<td>167, 175–177, 179</td>
</tr>
<tr>
<td>TAO.pidт</td>
<td>178</td>
</tr>
<tr>
<td><strong>messaging_init() operation</strong></td>
<td>510</td>
</tr>
<tr>
<td><strong>messaging_object() operation</strong></td>
<td>510</td>
</tr>
<tr>
<td>Microsoft Foundation Classes (MFC)</td>
<td>815, 1402</td>
</tr>
<tr>
<td><strong>MIF_Scheduling module</strong></td>
<td>246</td>
</tr>
<tr>
<td><strong>minimizing size</strong></td>
<td>1383</td>
</tr>
<tr>
<td>libraries</td>
<td>1397</td>
</tr>
<tr>
<td>minimum_corba build flag</td>
<td>182–183, 262, 282, 1378, 1383, 1385–1386, 1398</td>
</tr>
<tr>
<td><strong>See also footprint</strong></td>
<td></td>
</tr>
<tr>
<td><strong>minimum_corba build flag</strong></td>
<td>1378, 1383</td>
</tr>
<tr>
<td><strong>minor() operation</strong></td>
<td>106</td>
</tr>
<tr>
<td><strong>minPriority constant</strong></td>
<td>220</td>
</tr>
<tr>
<td><strong>MIOP</strong></td>
<td></td>
</tr>
<tr>
<td><strong>See multicast inter-ORB protocol (MIOP)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MMAPFilePrefix option</strong></td>
<td>384–385</td>
</tr>
<tr>
<td><strong>MMAPFileSize option</strong></td>
<td>384–385</td>
</tr>
<tr>
<td><strong>Model-Driven Architecture (MDA)</strong></td>
<td>xxxvii</td>
</tr>
<tr>
<td><strong>monolithic implementation</strong></td>
<td></td>
</tr>
<tr>
<td>CORBA Component Model (CCM)</td>
<td>1293, 1301, 1303</td>
</tr>
<tr>
<td><strong>most important first (MIF) scheduling</strong></td>
<td>246</td>
</tr>
<tr>
<td><strong>MPC</strong></td>
<td></td>
</tr>
<tr>
<td><strong>See MakeProjectCreator (MPC)</strong></td>
<td></td>
</tr>
<tr>
<td>mpc.pl program</td>
<td>34–35, 37–38, 41, 79</td>
</tr>
<tr>
<td><strong>MPC_COMMANDLINE environment variable</strong></td>
<td>42</td>
</tr>
<tr>
<td><strong>MPC_DEPENDENCY_COMBINED_STATIC_LIBRARY environment variable</strong></td>
<td>42</td>
</tr>
<tr>
<td><strong>MPC_LOGGING environment variable</strong></td>
<td>42</td>
</tr>
<tr>
<td><strong>MPC_SILENT environment variable</strong></td>
<td>42</td>
</tr>
<tr>
<td><strong>MPC_VERBOSE_ORDERING environment variable</strong></td>
<td>42</td>
</tr>
<tr>
<td><strong>msg_wfmo reactor</strong></td>
<td>737</td>
</tr>
</tbody>
</table>
multicast  624, 633, 635, 653, 665, 797, 818, 904, 911, 1052, 1062, 1416
  naming service  797, 800
  service discovery  635
  VxWorks  1416
multicast inter-ORB protocol (MIOP)  11, 374, 389, 391–393
multiplex receptacle
  CORBA Component Model (CCM)  1274–1277, 1297, 1299, 1310–1311
multithreading  527
  client thread  530
  connect strategies  529
    blocking  573
    leader-follower  572
    reactive  573
  data-centric publish-subscribe (DCPS)  1183
  example  557, 562, 565
  flushing strategies  529, 571–572, 575
  guideline  537, 542, 552
  main thread model  550
  multithreaded select reactor  569
  ORB-controlled model  548
  polling loop  537
  request invocation  530
  request processing  531
  select reactor  568
  server thread  530
  shutting server down  543
  single thread model  548
  thread-pool reactor  570
  transport multiplexing strategies  573
  wait strategies  529, 576
    wait-on-leader-follower  582
    wait-on-leader-follower-no-upcall  583
    wait-on-reactor  581
    wait-on-read  577
MutexNotFound exception  259
mwc.pl program  28–29, 35–38, 40–41, 44, 79

name() operation  294, 306
name_modifier option 38
NameServiceIOR environment variable 800, 828, 874
NameServicePort environment variable 665, 788, 799
   client source code 796
   command line options 818
   compliance 1443
   corbaloc object URL 801
   corbaname object URL 803
discriminating between multiple services 798
   example 794
      TAO_Naming classes 811, 823
executable 791
multicast 797, 800
NameServiceIOR environment variable 800, 828, 874
NameServicePort environment variable 665, 788, 799
NamingContextExt interface 807–808
NamingViewer utility 815
nsadd utility 814
nsdel utility 814
nslist utility 814
object URLs 801
options 818
   ORB_NameServicePort 624, 663, 665–666, 788, 799
ORB service libraries 784
   persistence 819
resolve_initial_references() operation 793
resolving 793
root naming context 828
TAO_Naming_Client class 810–811
TAO_Naming_Server class 811–812
TAO-specific classes 810
   utilities 814
      NamingViewer 815
      nsadd 814
      nsdel 814
      nslist 814

Naming_Service program 792
NamingContextExt interface 807–810
NamingViewer utility 815
negating the logic of filters
real-time event service (RTES) 891

nested upcalls 15, 579, 581–584

nesting groups
real-time event service (RTES) 892

NetworkPriorityMapping interface 256–258
NetworkPriorityMappingManager interface 257–259
new_for_consumers() operation 990, 1013
new_for_suppliers() operation 983, 1008
next() operation 475

NMake 19, 27, 36, 38, 44, 51
nmake build type 19, 27, 36, 38, 44, 51

no operation (NOOP) strategy 691, 696

no_context() operation 1126

NO_IMPLEMENT exception 355

node map 1290, 1323–1324, 1327, 1329, 1344, 1354–1355

NodeApplication program 1343, 1345, 1354–1355

NodeManager program 1329, 1343, 1345, 1354–1355

non_existent() operation 355

NON_RETAIN policy 356, 359

NOOP
See no operation (NOOP) strategy

noreldefs option 38, 44


calling
adding subscriptions 966
administration properties 963
architecture 950
collocated 1015
compatibility with event service 1021
conflict resolution 968
connecting
consumers 992
suppliers 993
ConnectionReliability property 962, 966, 970, 976
destroying the notification channel 994
developing structured event supplier 1006
disconnecting
consumers 992
suppliers 993
event consumer 1012
event filtering 954, 1001, 1003
consumer 1003
supplier 1001
event structure 951–952
event supplier 980, 1006
EventReliability property 962, 967, 972, 976, 1045
eexample 977
features 951
implementing interfaces
    sequence push consumer 1011
    sequence push supplier 1006
    structured push consumer 986
    structured push supplier 979
limits
    MaxConsumers 963, 967, 1004
    MaximumBatchSize 954, 961–962, 967
    MaximumEventsPerConsumer 961, 967, 1026
    MaxQueueLength 963–964, 967, 1004
    MaxSuppliers 963, 967, 1004
managing connections 992
MaxConsumers property 963, 967, 1004
MaxQueueLength property 963–964, 967, 1004
MaxSuppliers property 963, 967, 1004
negotiating quality of service (QoS) 968
notification administration properties 963
    MaxConsumers 963, 967, 1004
    MaxQueueLength 963–964, 967, 1004
    MaxSuppliers 963, 967, 1004
    RejectNewEvents 961, 963–964, 966–967
notify manager
    resource factory options 1022–1023, 1045
Notify_Service program 978, 982, 1015, 1021
obtain_subscription_types() operation 958
offer_change() operation 957
offers and subscriptions 994, 999
ORB service libraries 784
PacingInterval property 954, 961–962, 967
quality of service (QoS)
    get_qos() operation 968
    properties 1004
        accessing and modifying 966
        ConnectionReliability 962, 966, 970, 976
        DiscardPolicy 961, 963–964, 966–967, 1026
AnyOrder 961
DeadlineOrder 961
FifoOrder 961
LifoOrder 961
PriorityOrder 961
EventReliability 962, 967, 972, 976, 1045
MaximumBatchSize 954, 961–962, 967
MaximumEventsPerConsumer 961, 967, 1026
OrderPolicy 960, 967, 1005, 1026
AnyOrder 960
DeadlineOrder 961
FifoOrder 960
PriorityOrder 960
PacingInterval 954, 961–962, 967
PriorityOrder 960, 967
setting in a structured event header 967
StopTime 960, 967
Timeout 960, 967
support 959
quality of service (QoS) properties 959, 966, 1004
resuming consumer connections 993
sequence push consumer 1011
sequence push supplier 1006
setting quality of service (QoS) 967
structured event 952–953
consumer 988
supplier 980
type 952
structured push consumer 986–987
offer_change() operation 987
push_structured_event() operation 986
structured push supplier 979
subscriptions 956, 994, 999
adding to consumer 996
subscription_change() operation 957
suspending
consumer connections 993
supplier connections 993
TAO-specific properties
BlockingPolicy 966, 1026
ThreadPool 964, 967, 973–975, 1020, 1024
ThreadPoolLanes 965, 967, 973, 975, 1017, 1024
transmitting an EventBatch 1005
unsupported quality of service (QoS) exceptions 968

Notify_Service program 978, 982, 1015, 1021
command line options 1021
starting 978

notoplevel option 38
nsadd utility 814
nsdel utility 814
nslist utility 814
null filters
real-time event service (RTES) 894

O

O’Ryan, Carlos 1465–1467

object adapter
see portable object adapter (POA)

Object Computing, Inc. (OCI) i, xxvii, xxxii–xxxiii, xxxv, 3–4, 17–18, 1065, 1377, 1445

object factories 592–593

Object Management Group (OMG) specifications
C++ Language Mapping (formal/03-06-03) 12, 112, 1442–1443, 1466
CORBA 2.6 (formal/01-12-35) 146, 353, 1049, 1066, 1436
CORBA 3.0 (formal/02-06-33) 661, 786
CORBA 3.0.3 (formal/04-03-12) 12, 139–140, 146, 183, 282, 316–317, 372, 532, 658–659, 661, 802, 1049, 1066, 1436, 1466
CORBA Component Model (CCM) (formal/02-06-65) 353, 1213, 1253, 1288, 1364, 1445, 1466
Data Distribution Service for Real-Time Systems (formal/04-12-02) 1175–1176, 1445, 1466
Deployment and Configuration of Component-based Distributed Applications (ptc/03-07-08) 1287–1288, 1299, 1343, 1353, 1364, 1466
Event Service 1.2 (formal/04-10-02) 825, 949, 1466
Extensible Transport Framework (ETF) (ptc/04-03-03) 401, 1466
minimum CORBA (formal/02-08-01) 12
Naming Service 1.2 (formal/02-09-02) 791, 793, 1443, 1466
Notification Service 1.1 (formal/04-10-13) 949, 1444, 1467
Persistent State Service (formal/02-09-06) 1370, 1467
Real-Time CORBA 1.0 (ptc/99-06-02) 211
Real-Time CORBA 1.1 (formal/02-08-02) 211
Real-Time CORBA 1.2 (formal/05-01-04) 12, 211, 1441, 1467
Real-Time CORBA 2.0 (dynamic scheduling) (formal/03-11-01) 211
Security Service (original version) (security/00-12-02) 1065
Security Service 1.8 (formal/02-03-11) 1068

**object reference**
See interoperable object reference (IOR)

**object URL** 801–807
- corbaloc 365–366, 786–787, 800–803, 808, 815
- corbaname 700, 720, 803–804, 806–807, 810, 814

**object_key_delimiter() operation** 482

**OBJECT_NOT_EXIST exception** 103, 358, 542

**object_to_string() operation** 26

**objects by value (OBV)**
See value types

**observers**
- custom gateways 902
- real-time event service (RTES) 901

**obtain_notification_push_consumer() operation** 984, 1008–1009

**obtain_notification_push_supplier() operation** 990, 1013

**obtain_offered_types() operation** 958, 996, 999–1000

**obtain_subscription_types() operation** 958, 999–1000

**ObtainInfoMode interface** 999–1000

**OBV**
See value types

**OCIReleaseNotes.html file** xxix

**OctetSeq** 300–301, 304

**offer_change() operation** 957, 987–988, 994–995, 999–1000, 1011–1012

**OMG**
See Object Management Group (OMG) specifications

**on_data_available() operation** 1193

**on_liveliness_changed() operation** 1198

**oneway invocation** 14, 140–141, 175–179, 221, 234, 332, 386–387, 389, 392, 531, 544, 579, 649

**open source software** xxv

**open() operation** 468

**open_named_mutex() operation** 259

**OpenSSL** 374, 1068, 1088, 1096, 1099–1101, 1103, 1105–1108, 1110–1111, 1125

**OPENSSL_CONF environment variable** 1099–1100, 1106

**operating systems**
- Linux xxxiv, 21, 507, 1107
- LynxOS 1423, 1425
- QNX 380
VxWorks 630, 811
Windows 18, 21–22, 24, 29, 77, 97, 99, 611, 815, 1107–1108, 1369

**operation lookup**
- binary search 95
- dynamic search 95
- linear search 95
- perfect hash 95

**operations**
- _add_ref() 322, 363, 1239
- _cxx_register() 1189
- _get_policy() 268
- _id() 104, 106–107, 109, 157, 313, 391, 831–832
- _narrow() 367
- _refcount_value() 322
- _remove_ref() 322, 363, 1239, 1256
- _rep_id() 106–107, 109
- _retn() 1247
- _set_policy_overrides() 170
- _unchecked_narrow() 339–340, 392
- _validate_connection() 174, 228
- accept_svc_handler() 423
- activate() 26, 298, 567, 742, 794, 812–813, 992, 1016, 1131, 1253–1254, 1257, 1259, 1279
- activate_object() 26
- activate_object_with_id() 157, 391
- activate_object_with_id_and_priority() 271
- activate_object_with_priority() 271
- activate_svc_handler() 423
- add_client_request_interceptor() 292
- add_constraints() 1002
- add_filter() 1002
- add_iior_component() 308
- add_iior_interceptor() 307–308
- add_server_request_interceptor() 297
- add_transport_to_cache() 472
- addr_to_string() 475
- amh_response_handler_allocator() 704
- ami_response_handler_allocator() 704
- begin_scheduling_segment() 236, 238, 273, 275
- bind() 366
cache_maximum() 696–697
cancel() 236
ccm_activate() 1253–1254, 1257, 1259–1260, 1268, 1271, 1279–1280
ccm_passivate() 1253, 1259–1260, 1268, 1271
ccm_remove() 1253, 1259, 1268, 1271
ciaopostactivate() 1253–1254, 1260, 1268, 1271
ciao_preactivate() 1253–1254, 1260, 1268, 1271
close_connection() 470, 504
codecfactory() 300, 304
connect_push_consumer() 837, 881, 886, 901
connect_push_supplier() 829, 837, 876, 901
connect_sequence_push_consumer() 1014
connect_sequence_push_supplier() 1009
connect_structured_push_consumer() 991
connect_structured_push_supplier() 985
connection_handler_i() 510
consolidate_message() 508
create() 1225, 1263–1264
createCachedConnectionLock() 696–697
create_channel() 982, 1016
create_corba_object_lock() 705
create_filter() 1001
create_flushing_strategy() 695
create_for_unmarshal() 324
create_ft_service_retention_id_lock() 764
create_lk_strategy() 692
create_mutex() 223
create_named_mutex() 259
create_participant() 1187
create_POA() 171, 215, 219, 224
create_policy() 168–169, 172, 174, 176, 179, 182, 1121–1124
create_priority_banded_connection_policy() 228
create_proxy() 332, 336–337, 339–340, 342–343, 347
create_purging_strategy() 696–697
create_reference_with_id_and_priority() 271
create_reference_with_priority() 271
create_resource_manager() 240, 242
decode_profile() 481
default_filter_factory() 1001
delete_contained_entities() 1194
destroy() 26, 534
destroy_mutex() 223
disconnect_push_consumer() 831, 837, 843, 878, 901
disconnect_push_supplier() 829–830, 837, 876, 901
disconnect_sequence_push_consumer() 1011
disconnect_sequence_push_supplier() 1006
disconnect_structured_push_consumer() 986–988
disconnect_structured_push_supplier() 979
do() 237, 273
duplicate() 475
encode_endpoints() 486
end_scheduling_segment() 238, 273
endpoint() 485
endpoint_count() 486
establish_components() 306
event_handler_i() 510
find_topic() 1190
fini() 493, 608, 610
generate_request_header() 512
get_client_policy() 355
get_connections_content() 1283
get_domain_managers() 355
get_effective_component() 307, 309
get_event_channel() 983, 989
get_interface() 355
get_parser_names() 700
get_peer_certificate() 1125
get_peer_certificate_chain() 1125
get_policy() 355, 1077
get_policy_overrides() 355
get_protocol_factories() 699
get_proxy() 337–338
get_qos() 964, 968
get_slot() 311
handle_events() 518, 569–570, 581
handle_input() 410–411, 414, 423–424, 467, 471, 508, 518, 524, 570
handle_output() 470
handle_timeout() 471, 545
hash() 355, 476, 482
id_to_reference() 26
id_to_reference() 409
info() 614, 616
init() 493–494, 608, 810–811
init_protocol_factories() 699
input_cdr_allocator_type_locked() 694
input_cdr_buffer_allocator() 693
input_cdr_dblock_allocator() 693
input_cdr_msgblock_allocator() 693
instance() 338, 599, 615
invoke() 511
is_a() 355, 367
is_equivalent() 355, 474
is_nil() 27, 541
locate() 368
lock() 222, 240
locked_transport_cache() 696–697
make_acceptor() 494
make_connector() 495
make_mprofile() 483
make_svc_handler() 423
mapping() 251
match_prefix() 495–496, 700
max_muxed_connections() 696–697
messaging_init() 510
messaging_object() 510
name() 294, 306
new_for_consumers() 990, 1013
new_for_suppliers() 983, 1008
next() 475
no_context() 1126
non_existent() 355
object_key_delimiter() 482
object_to_string() 26
obtain_notification_push_consumer() 984, 1008–1009
obtain_notification_push_supplier() 990, 1013
obtain_offered_types() 958, 996, 999–1000
obtain_subscription_types() 958, 999–1000
offer_change() 957, 987–988, 994–995, 999–1000, 1011–1012
on_data_available() 1193
on_liveliness_changed() 1198
open() 468
open_named_mutex() 259
options_delimiter() 496
output_cdr_buffer_allocator() 694
output_cdr_dblock_allocator() 694
output_cdr_msgblock_allocator() 694
parse_string() 485, 701
parse_string_i() 482–483
perform_work() 154–155, 159, 534, 536–541, 544, 546, 550, 553, 562, 579, 582
ping() 1139–1140
post_init() 286, 291–292, 297, 307–308
postinvoke() 356–358
pre_init() 286, 291, 307
prefix() 496
preinvoke() 356–357
process_directive() 602
pthread_create() 567
purge_percentage() 696–697
push() 829, 873, 878–880, 891
push_message_consumer() 1268, 1270
push_structured_event() 985–987, 992
push_structured_events() 1010–1011, 1015
rebind() 366, 845
receive_exception() 283
receive_other() 283
receive_reply() 283
receive_request() 283, 261, 284
receive_request_service_context() 284
reclaim_reactor() 692
reconfigure() 602, 616
recv() 509, 524
reference_to_servant() 1195
register() 1189
register_handler() 509
register_new_factory() 324
register_proxy_factory() 336–337
register_value_factory() 319
release() 363, 541
requires_explicit_endpoint() 497
resolve() 828
resource_usage_strategy() 706–707
resume_connection() 993
resume_handler() 470, 472
RTScheduling::Scheduler::create_resource_manager() 240, 242
RTScheduling::Scheduler::set_scheduling_parameter() 242
run() 26, 159, 406, 409, 532, 537, 543, 550, 563, 582, 811, 1165
select() 403, 569–570, 737
send() 504
send_exception() 284
send_message() 506
send_message_shared() 507, 524
send_message_shared_i() 512, 523
send_other() 285
send_poll() 283
send_reply() 284
send_request() 283, 505
send_synchronous_message_i() 516
set_locator() 368–369
set_policy_overrides() 170, 355, 1076, 1083, 1087
set_qos() 964
set_scheduling_parameter() 242
set_session_context() 1253–1254, 1259, 1268, 1270–1271, 1279
set_slot() 311
shutdown() 534, 541–546, 1128–1129, 1131, 1139, 1164
spawn() 235, 237–238, 273
string_dup() 27
subscription_change() 957–958, 979–980, 991, 996–997, 999–1000, 1006, 1014
suspend_connection() 993
svc() 276, 508, 567, 811–812, 1254, 1256–1257, 1259, 1275, 1278–1281
TaggedComponent() 309
take() 1196
take_instance() 1196
take_next_instance() 1196
take_next_sample() 1196
target_is_a() 295
the_POAManager() 26
to_CORBA() 221, 248, 256, 258
to_native() 221, 248
to_network() 256, 258
to_string() 485
to_url() 808
try_lock() 223, 240
unbind() 366
unlock() 222–223, 240
unregister_proxy_factory() 336, 343
update_scheduling_segment() 239
use_locked_data_blocks() 694
validate_connection() 355
WaitForMultipleObjects() 403, 737
work_pending() 154, 534, 536–538, 540–541, 544

operations handle_close() 467
optimize build flag 1378, 1381

options
BUFFERSIZE_COUNT 178–179
BUFFERSIZE_TIMEOUT 178
CECConsumerControl 849, 853–855, 861, 868
CECConsumerControlPeriod 849, 854–855, 861, 867–868
CECConsumerControlTimeout 849–850
CECDispatching 847–848, 851, 856–857
CECDispatchingThreads 847–848, 856–857
CECProxyConsumerCollection 850, 858–859
CECProxyConsumerLock 848, 860
CECProxyDisconnectRetries 849
CECProxySupplierCollection 850, 862–863
CECProxySupplierLock 848, 864
CECReactivePullingPeriod 852, 865
CECSupplierControl 849, 866–867
CECSupplierControlPeriod 849, 867
CECSupplierControlTimeout 849–850
CECUseORBId 852, 869
ECDDispatching 915–916, 920, 924–926
ECD dispatchingThreads 916, 924, 926
ECFFiltering 887, 892, 915, 917, 927
ECObserver 901, 917, 928
ECProxyConsumerLock 918, 929
ECProxySupplierLock 918, 934
ECScheduling 917, 927, 935
ECSupplierFilter 917
ECSupplierFiltering 917, 938
ECTimeout 916, 939
ECUseORBId 917, 940
ORBSchedPolicy 263–264
ORBScopePolicy 263, 265
SO LINGER 660
SSLAuthenticate 1134
SSLCertificate 1133
SSLNoProtection 1132–1133
SSLPrivateKey 1132, 1134
TCP_NODELAY 669
tp 737

**options_delimiter() operation** 496

**ORB** 593, 620, 740–741
configuring real-time CORBA 263
connection management 623
controlling service configurator 621
core 11, 15
debugging 622
dendsystem 8
event handling 532, 534–540
initialization 828
initialization options 617–688
interface definition 618
Java xxxiv–xxxv
mediation 354
optimizing request processing 622
options

ORBCDRTradeoff 622, 626, 713
ORBColocation 587, 618, 623, 627
ORB Daemon 597, 621, 630, 662
ORBDebug 618, 620, 622, 631
ORBDebugLevel 620, 622, 632, 1114
ORBDefaultInitRef 618, 623, 633–636, 638–639, 654–655, 664, 787, 801, 803, 806
ORBDisableRTCollocation 266, 623, 640–641
ORBDottedDecimalAddresses 377, 624, 642–643
ORBEndpoint 405, 408, 620, 624, 642, 644–651, 658, 800
ORBEnforcePreferredInterfaces 652
ORBId 618–620, 852, 917
ORBImplRepoServicePort 624, 653, 788
ORBLaneEndpoint 227, 624, 658–659
ORBLaneListenEndpoints 227, 624, 659
ORBLingerTimeout 660
ORBMulticastDiscoveryEndpoint 788, 799
ORBNameServicePort 624, 663, 665–666, 788, 799
ORBNegotiateCodesets 623, 667–668
ORBDelay 623, 669
ORBNoProprietaryActivation 618, 670
ORBObjRefStyle 622, 671–672, 1153
ORBPREFERREDINTERFACES 652, 673–674
ORBRcvSock 618, 623, 675
ORBServerId 618, 625, 676
ORBServiceConfigLoggerKey 621, 677
ORBSingleReadOptimization 678–679
ORBSndSock 623, 680
ORBStdProfileComponents 623, 681
ORBSvcConf 598, 621, 682, 847, 915, 941, 1116–1117
ORBSvcConfDirective 598, 621, 683
ORBTradingServicePort 624, 684, 788
ORBUseIMR 625, 685, 1144, 1147, 1157
ORBUseSharedProfile 687
ORBVerboseLogging 688

protocol selection 623
protocols 699
real-time 217, 221, 256, 263
service libraries 642, 783–784
  audio/video streaming 13, 783
  concurrency control 13, 784
  data distribution 13
  event 13, 784
  interface repository 13
  life cycle 13, 784
  load balancing 13, 784
  log 13, 784
  naming 13, 784
  notification 13, 784
  property 13, 784
  real-time event 14, 784
  real-time scheduling 784
  security 13, 784
  time 13, 784
  trading 13, 784
shutdown 543–546
wait_for_completion parameter 534, 541–545

**ORB_CTRL_MODEL policy** 547, 551–552, 554
<table>
<thead>
<tr>
<th>Parameter/Option</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>orb_identifier parameter</td>
<td>618–619</td>
</tr>
<tr>
<td>ORBActiveHintInIDs option</td>
<td>747, 749</td>
</tr>
<tr>
<td>ORBActiveHintInPOANames option</td>
<td>746, 750</td>
</tr>
<tr>
<td>ORBActiveObjectMapSize option</td>
<td>747, 751</td>
</tr>
<tr>
<td>ORBAllowReactivationOfSystemIDs option</td>
<td>748–749, 752</td>
</tr>
<tr>
<td>ORBAMHResponseHandlerAllocator option</td>
<td>705, 731</td>
</tr>
<tr>
<td>ORBAMIResponseHandlerAllocator option</td>
<td>705, 732</td>
</tr>
<tr>
<td>ORBCDRTadeoff option</td>
<td>622, 626, 713</td>
</tr>
<tr>
<td>ORBCCharCodesetTranslator option</td>
<td>704, 714</td>
</tr>
<tr>
<td>ORBCClientConnectionHandler option</td>
<td>768</td>
</tr>
<tr>
<td>ORBCollocation option</td>
<td>587, 618, 623, 627</td>
</tr>
<tr>
<td>ORBConcurrency option</td>
<td>556, 558, 560, 562, 742, 753</td>
</tr>
<tr>
<td>ORBConnectionCacheLock option</td>
<td>694, 698, 715, 734</td>
</tr>
<tr>
<td>ORBConnectionCacheMax option</td>
<td>698, 716</td>
</tr>
<tr>
<td>ORBConnectionCachePurgePercentage option</td>
<td>698, 717, 733</td>
</tr>
<tr>
<td>ORBConnectionPurgingStrategy option</td>
<td>697, 712, 733</td>
</tr>
<tr>
<td>ORBCorbaObjectLock option</td>
<td>706</td>
</tr>
<tr>
<td>ORBDaemon option</td>
<td>597, 621, 630, 662</td>
</tr>
<tr>
<td>ORBDebug option</td>
<td>618, 620, 622, 631</td>
</tr>
<tr>
<td>ORBDebugLevel option</td>
<td>620, 622, 632, 1114</td>
</tr>
<tr>
<td>ORBDefaultInitRef option</td>
<td>618, 623, 633–636, 638–639, 654–655, 664, 787, 801, 803, 806</td>
</tr>
<tr>
<td>ORBDisableRTCcollocation option</td>
<td>266, 623, 640–641</td>
</tr>
<tr>
<td>ORBDottedDecimalAddresses option</td>
<td>377, 624, 642–643</td>
</tr>
<tr>
<td>ORBEndpoint option</td>
<td>405, 408, 620, 624, 642, 644–651, 658–659, 800</td>
</tr>
<tr>
<td>ORBEnforcePreferredInterfaces option</td>
<td>624, 652</td>
</tr>
<tr>
<td>ORBFflushingStrategy option</td>
<td>576, 580, 695, 719</td>
</tr>
<tr>
<td>blocking strategy</td>
<td>719</td>
</tr>
<tr>
<td>leader_follower strategy</td>
<td>719</td>
</tr>
<tr>
<td>reactive strategy</td>
<td>719</td>
</tr>
<tr>
<td>ORBId option</td>
<td>618–620, 852, 917</td>
</tr>
<tr>
<td>ORBImplRepoServicePort option</td>
<td>624, 653, 788</td>
</tr>
<tr>
<td>ORBInitializer interface</td>
<td>286, 291, 297, 307, 313</td>
</tr>
<tr>
<td>ORBInitInfo interface</td>
<td>286</td>
</tr>
<tr>
<td>ORBInitRef option</td>
<td>496, 618, 624, 633–636, 638, 654–657, 664, 666, 671, 685, 1144, 1147, 1156, 1165, 1167</td>
</tr>
<tr>
<td>ORBInputCDRAlocator option</td>
<td>558–559, 694–695, 712, 715, 734</td>
</tr>
</tbody>
</table>
ORBIORParser option 702–703, 720
ORBLaneEndpoint option 227, 624, 658–659
ORBLaneListenerEndpoints option 227, 624, 659
ORBLingerTimeout option 624, 660
ORBLogFile option 622, 662
ORBMulticastDiscoveryEndpoint option 624, 663, 788, 799
ORBMuxedConnectionMax option 698, 721
ORBNameServicePort option 624, 663, 665–666, 788, 799
ORBNativeCharCodeset option 704
ORBNativeWCharCodeset option 704
ORBNegotiateCodesets option 623, 667–668
ORBNetworkPriorityMapping option 257, 264
ORBndelay option 623, 669
ORBNoProprietaryActivation option 618, 625, 670
ORBObjectKeyTableLock option 699, 724
ORBObjRefStyle option 622, 671–672, 1153
ORBPersistentIDPolicyDemuxStrategy option 746, 754
ORBPOLock option 558, 742, 755
ORBPOLockType option 742
ORBPOLMapSize option 746, 756
ORBPolicyManager option 170, 172
ORBPreferredInterfaces option 624, 652, 673–674
ORBPriorityMapping option 251, 263
ORBProfileLock option 558, 765, 771
custom factory 725
DIOP_Factory 725
IIOP_Factory 725
IIOP_Lite_Factory 725
SHMIOP_Factory 725
SSLIOP_Factory 725
UIOP_Factory 725
UIOP_Lite_Factory 725
ORBRecSock option 618, 623, 675
ORBReactorMaskSignals option 727
ORBReactorThreadQueue option 712, 735–736
ORBReactorType option 557–559, 568–570, 683, 693, 712, 737–738
ORBResourceUsage option 707, 728
ORBSchedPolicy option 263–264
ORBScopePolicy option 263, 265
ORBServerId option 618, 625, 676
ORBServiceConfigLoggerKey option 598, 621, 677
ORBSingleReadOptimization option 623, 678–679
ORSndSock option 623, 680
ORBStdProfileComponents option 623, 681
ORBSvcConf option 598, 621, 682, 847, 915, 941, 1116–1117
ORBSvcConfDirective option 598, 621, 683
ORBSystemIDPolicyDemuxStrategy option 748, 757
ORBThreadFlags option 743, 758
ORBThreadPerConnectionTimeout option 561–562, 743, 759
ORBTradingServicePort option 624, 684, 788
ORBTransientIDPolicyDemuxStrategy option 747, 760
ORBTransportMuxStrategy option 561, 574, 580, 766, 772
ORBUniqueIDPolicyReverseDemuxStrategy option 748, 761
ORBUseIMR option 625, 685, 1144, 1147, 1157
ORBUserIDPolicyDemuxStrategy option 748, 762
ORBUseSharedProfile option 624, 687
ORBVerboseLogging option 622, 688
ORBWCharCodesetTranslator option 704, 729
OrderPolicy property 960, 967, 1005, 1026
Othman, Ossama 1466
output_cdr_buffer_allocator() operation 694
output_cdr_dblock_allocator() operation 694
output_cdr_msgblock_allocator() operation 694

PacingInterval property 954, 961–962, 967
package configuration descriptor 1289, 1323–1327, 1330
parse_string() operation 485, 701
parse_string_i() operation 482–483
PATH environment variable 597, 1106, 1294, 1332, 1349, 1352, 1382, 1392–1393, 1403, 1410–1411, 1419, 1421
Pattern Languages of Program Design 3: Software Patterns Series book 1466
Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2) book 16, 404, 563, 570, 582, 592, 735, 1467

patterns 15
abstract factory 15, 404
acceptor 15, 404
active object 15, 742
adapter 15
asynchronous completion token (ACT) 157
bridge 15
component configurator 404
composite 1286
connector 15, 404, 511
half-sync/half-async 15
leader/followers 563
manager 404
optimization principle 1467
proxy 15
reactor 15, 404, 689, 1467
service configurator 15, 404, 701
strategy 15, 404
template method 404, 508, 511, 513, 520, 524
thread-specific storage (TSS) 15

perfect hashing 10, 15, 95, 744
See also gperf

perform_work() operation 154–155, 159, 534, 536–541, 544, 546, 550, 553, 562, 579, 582

per-hop-behavior (PHB) 252
PHB
See per-hop behavior (PHB)

PICurrent interface 284, 311–314, 316
ping() operation 1139–1140
pipes build flag 1379, 1386
platform independent model (PIM) 1175
platform specific model (PSM) 1175
platform support xxviii, 5
platforms supported xxxiii
acceptor 416, 418, 689
acceptor typedefs 420
accept-strategy template 425
address definition 377, 381, 385, 387–388, 390
building 375
client invocation 411
combining protocols 393
concurrency-strategy template 426, 451
configuring SHMIOP factory 384
connection
  handler 417, 464, 689
  request 410
connector 416, 441, 689
connector typedefs 444
connect-strategy template 450
constructor 474
creation-strategy template 424, 449
declaring 376, 380
developing 401
dynamic loading 491
dynamic loading 491
endpoint 417, 473
example 405, 411
framework 414
guidelines 382
identifier tags 403
loading 375, 383, 386, 388, 390
member functions 427, 453, 467, 480, 493, 503
pluggable-acceptor
  close() operation 433
  constructor 427
  create_profile() operation 433
  destructor 428
  endpoint_count() operation 438
  is_collocated() operation 437
  object_key() operation 439
  open() operation 428
  open_default() operation 431
pluggable-connect
  check_prefix() operation 462
pluggable-connection-handler
  add_transport_to_cache() operation 472
  constructor 467
handle_input() operation 471
open() operation 468
resume_handler() operation 472
pluggable-connector
close() operation 455–456
connect() operation 457
constructor 453
create_profile() operation 461
destructor 454
make_profile() operation 463
object_key_delimiter() operation 463
open() operation 454
pluggable-endpoint
addr_to_string() operation 475
duplicate() operation 475
hash() operation 476
is_equivalent() operation 474
next() operation 475
pluggable-profile
constructor 480
decode() operation 481
encode() operation 482
hash() operation 482
object_key_delimiter() operation 482
to_string() operation 485
pluggable-protocol-factory
constructor 493
init() operation 494
make_connector() operation 495
match_prefix() operation 495
options_delimiter() operation 496
prefix() operation 496
requires_explicit_endpoint() operation 497
pluggable-transport
close_connection() operation 504
constructor 503
event_handler_i() operation 510
handle_input_i() operation 508
messaging_init() operation 510
messaging_object() operation 510
recv_i() operation 509
send_i() operation 504
send_message() operation 506
send_message_shared() operation 507
send_request() operation 505
profile 417, 477
protocol factory 486, 689
protocol-specific
  parse_options() operation 440
requirements 402
restrictions 387, 392
strategy acceptor template 422
strategy connector template 446
TAO_Acceptor class 416, 418
TAO_Connector class 416
TAO_Endpoint class 417
transport 373, 497, 511
transport behavior 403
using ACE_wrappers 525

POA
  See portable object adapter (POA)

poa_policies attribute 242

policies
  BidirectionalPolicyValue 181
  BufferingConstraint 177–179
  ConnectionTimeout 174
  Durability 1200
  Entity 1200
  History 1199
  Liveliness 1197
  MAIN_THREAD_MODEL 547, 550–551, 553–554
  NON_RETAIN 356, 359
  ORB_CTRL_MODEL 547, 551–552, 554
  Presentation 1200
  PriorityModelPolicy 228
  PrivateConnectionPolicy 230
  PROCESS 265
  Reliability 1198
  RequestProcessingPolicy 359
  ResourceLimits 1199
  RETAIN 356, 477, 1380
  SCHED_FIFO 264
  SCHED_OTHER 264
  SCHED_RR 264
scope_policy 265
ServantRetentionPolicy 356, 359
SERVER_DECLARED 269
SINGLE_THREAD_MODEL 547, 551–553
SyncScope 167, 175–177, 179
SYSTEM 265
ThreadPolicy 547, 550–551, 554
USE_SERVANT_MANAGER 359

Policy interface 356
PolicyCurrent interface 157, 170, 355, 654
PolicyManager interface 170, 172, 176, 179, 254, 355
Pollable interface 355
PollableSet interface 355–356
port
   CORBA Component Model (CCM) 1215–1216, 1224, 1265, 1291, 1293, 1297, 1299–1301, 1307, 1310–1311, 1317, 1319, 1329, 1354
portability 15
   asynchronous method handling (AMH) 200
   client interceptors 282–283, 288–289, 301
   ClientInterceptor class 288
   ClientRequestInfo interface 285
      get_effective_component() 307
   codec (coder/decoder) 300
   codec_factory() operation 300, 304
   CodecFactory reference 300–301
   coders 316
   context_data field 290
   context_id field 290
   decoders 316
   decoding the tag 309
   destroy() operation 306
   example 287, 307
      client-side recursion 312
      simple authentication interceptor 287
      using the codec 300
   extracting tagged information 307
   get_slot() operation 311
   installing the interceptor 291, 297
interceptors build flag  629, 1074, 1077
IOR interceptors  305–308, 316
marshaling  300
name() operation  294, 306
ORBInitializer interface  291
PortableInterceptor.pidl  284–285, 305
PortableInterceptor::Current interface  311
PortableInterceptor::IORInfo interface  306
PortableInterceptor::PICurrent interface  284, 311–314, 316
PortableServer module  298
post_init() operation  286
pre_init() operation  286
real-time CORBA  235, 240, 260–261
receive_exception() operation  283
receive_other() operation  283
receive_reply() operation  283
receive_request() operation  284
receive_request_service_context() operation  284
registering interceptors  286
  IOR  307
request interceptors  282
request parameters  285
send_exception() operation  284
send_other() operation  285
send_poll() operation  283
send_reply() operation  284
send_request() operation  283
server interception points  284
server request interceptors  284, 293–294, 304
server-side interceptor  293
service context  235, 299
set_slot() operation  311
tagged components  305
TAO_Local_RefCounted_Object class  306
target_is_a() operation  295

**portable object adapter (POA)**  10, 12
active object map  157, 356, 744, 747, 751, 761
manager  628, 740, 835, 880, 899
map  744
map options  746
policy
  MULITPLE_ID  761
PERSISTENT 754, 756
SYSTEM_ID 751
TRANSIENT 756
UNIQUE_ID 761
USER_ID 749, 751

**portable priorities** 14, 213

**PortableServer module** 330, 391
  interfaces 355
  PortableServer::ServantActivator interface 356
  PortableServer::ServantLocator interface 356–357, 359

**portspan option** 378–379, 388, 645–646, 650–651

**POSA2**
  See *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2)* book

**post_init() operation** 286, 291–292, 297, 307–308

**postinvoke() operation** 356–358

**pre_init() operation** 286, 291, 307

**prefix filter builder**
  real-time event service (RTES) 887

**prefix() operation** 496

**preinvoke() operation** 356–357

**priority**
  end-to-end 14, 213, 227
  model 269, 272
    client-propagated 268
    server-declared 272
  timers
    real-time event service (RTES) 873


**priority mapping** 222, 247, 250
  continuous 247
  custom network 258
  direct 248
  linear 249
  maxPriority 220
  minPriority 220
  network 252–255, 257–258
  NetworkPriorityMapping interface 256–258
  NetworkPriorityMappingManager interface 257–259
  ORBNetworkPriorityMapping option 257, 264
  ORBPriorityMapping option 251, 263–264
ORBPriorityRange option 217
PriorityMapping interface 221, 251
PriorityMappingManager interface 251

**priority model**
- client-propagated 14, 266
- server-declared 14

**priority propagation** 227–228
- end-to-end 14, 213, 227

**priority-banded connections** 228

**PriorityBands** 229
**PriorityMapping interface** 221–222, 251
**PriorityMappingManager interface** 251

**PriorityModelPolicy policy** 228, 268

**private keyword**
- value types 320, 322

**PrivateConnectionPolicy policy** 230

**probe build flag** 1378, 1383

**process composition**
- CORBA Component Model (CCM) 1229

**process keyword**
- CORBA Component Model (CCM) 1231

**process_directive() operation** 602, 616

**profile build flag** 1378, 1383

**profile locking** 764

**properties**
- BlockingPolicy 966, 1026
- ConnectionReliability 962, 966, 970, 976
- DiscardPolicy 961, 963–964, 966–967, 1026
- EventReliability 962, 967, 972, 976, 1045
- MaxConsumers 963, 967, 1004
- MaxEventsPerConsumer 961, 967, 1026
- MaximumBatchSize 954, 961–962, 967
- MaxQueueLength 963–964, 967, 1004
- MaxSuppliers 963, 967, 1004
- OrderPolicy 960, 967, 1005, 1026
- PacingInterval 954, 961–962, 967
- Priority 960, 967
- RejectNewEvents 961, 963–964, 966–967
- ThreadPool 964, 967, 973–975, 1020, 1024
- ThreadPoolLanes 965, 967, 973, 975, 1017, 1024
- Timeout 960, 967

**property service** 13
ORB service libraries 784

**protocol**

asymmetric 393–394

**ProtocolList** 232

**protocols** 649

corbaloc rir 801

CSI ECMA 1075, 1079
datagram inter-ORB protocol (DIOP) 11, 374, 385–387, 494, 497, 644, 649–650, 725

DCE Common Inter-ORP protocol (DCE-CIOP) 1075, 1078

environment specific inter-ORB protocol (ESIOP) 371–372

factories 417, 699, 726


  lite 373

generic security service (GSS) 1079

HTTP tunneling bi-directional protocol (HTBP) 393

HTTP tunneling inter-ORB protocol (HTIOP) xxviii, 11, 374, 393

hypertext transfer protocol (HTTP) 11


  lite 374–375, 379–380, 725–726

  options 378

lite 726

magic number 379

multicast inter-ORB protocol (MIOP) 11, 374, 389, 391–393

pluggable xxviii, 5, 11, 15

properties 213, 230

provided with TAO 374

resolve initial reference 634

SCTP inter-ORB protocol (SCIOP) xxviii, 11, 374, 396

secure inter-ORB protocol (SECIOP) 1075, 1078–1079

secure socket layer inter-ORB protocol (SSLIOP) 11, 374, 388, 645, 649, 1065, 1086–1087, 1093, 1106, 1108, 1111, 1114–1118, 1120, 1122–1135, 1445

secure sockets layer (SSL) 11, 374, 388, 649, 1067, 1079, 1087

shared memory inter-ORB protocol (SHMIOP) 11, 374, 382–385, 393, 399, 403, 415–417, 432, 482, 494, 497, 634, 644, 648, 651, 655–657, 692, 725, 767, 801, 808

stream control transmission protocol (SCTP) 11, 396

transmission control protocol/internet protocol (TCP/IP) 9, 231, 371–373, 376, 383, 403, 642, 656, 699, 725, 913, 1093

unreliable IP multicast protocol (UIPMC) 374–375, 389–391, 393

protocols properties 231

provides keyword
CORBA Component Model (CCM) 1214, 1224, 1227, 1299

proxy collection options
CECProxyConsumerCollection 850, 858–859
CECProxySupplierCollection 850, 862–863
ECProxyPushConsumerCollection 919
ECProxyPushSupplierCollection 919
real-time event service (RTES) 919

proxy factory adapter 336

proxy pattern 15

ProxyConsumer interface 955, 984, 999, 1009
ProxyPushSupplier interface 833
ProxySupplier interface 955, 990, 999, 1013–1014
pseudo-IDL (PIDL) 173, 178, 217, 282, 284–285, 305, 311, 353, 1386

pthread_create() operation 567

public keyword
value types 320, 322

public-key cryptography 1089

publish and subscribe paradigm 826
consumers 826
suppliers 826

publisher

publishes keyword
CORBA Component Model (CCM) 1214, 1224, 1227, 1299

publish-subscribe 1176

pull model support 826, 838

Pure CORBA book xxvii, xxxii, 317, 330, 792, 1465

purge_percentage() operation 696–697

purging strategies 696
first in first out (FIFO) 691, 696
least frequently used (LFU) 691
least recently used (LRU) 691
no operation (NOOP) 691, 696

**purify build flag** 1378, 1382

**push consumer interface**
real-time event service 878

**push() operation** 829, 873, 878–880, 891
**push_message_consumer() operation** 1268, 1270
**push_structured_event() operation** 985–987, 992
**push_structured_events() operation** 1010–1011, 1015

**PushConsumer interface** 878, 880
  
  disconnect_push_consumer() operation 831, 878
  push() operation 878

**PushSupplier interface** 830, 876

Pyarali, İrfan 1467

---

**Q**

**QNX operating system** 380

**QoP**
See quality of protection (QoP)

**Qt GUI toolkit** 690, 707, 711

**qt_reactor build flag** 1378, 1384

**quality of protection (QoP) policy** 1118

  
  asynchronous requests 177
  buffered oneway 177
  connection timeouts 173
  data-centric publish-subscribe (DCPS) 1179–1180, 1183, 1187, 1189, 1196
  notification service 959
  object reference level 170
  oneway 177
  ORB level 170
  ORBPolicyManager interface 170, 172
  policies 168
  policy framework 215
  properties
    
    notification service 1004

**RELATIVE_RT_TIMEOUT_POLICY_TYPE** 172
reliable oneway calls 175
request and reply timeouts 171
support 959
SyncScope 175, 177
  SYNC_NONE 175
  SYNC_WITH_SERVER 175
  SYNC_WITH_TARGET 176
  SYNC_WITH_TRANSPORT 175
thread level 170
time independent invocation (TII) 139
quantify build flag 1071, 1378, 1382

rate-monotonic scheduling
See real-time CORBA, fixed priority scheduling

Rational Software
Purify 1382
Quantify 1382
reactive
  concurrency model 427, 554–558, 564
  strategy 719
reactors 15–571, 689, 1467
  ACE_QtReactor class 692, 707–708
  ACE_XtReactor class 692, 709
  fl_reactor build flag 1378, 1384
  gtk_reactor build flag 1378, 1384
  management 692
  msg_wfmo option 737
  qt_reactor build flag 1378, 1384
  reclaim_reactor() operation 692
  select_mt option 569–570, 712, 737
  select_st option 557–559, 568–569, 692, 712, 737
  thread pool 225
  tk_reactor build flag 1378, 1384
  tk_reactor option 712, 737
  wfmo option 540, 712, 737
  xt_reactor build flag 1378, 1383
real-time collocation resolver 265, 587, 640
real-time CORBA 10, 14, 64, 135, 211–234, 246, 250–259, 262–272, 400, 474, 554, 561, 565, 1441, 1467
   _get_policy() operation 268
architecture 214–215
asynchronous invocations 234
attributes
   poa_policies 242
   scheduling_discipline_name 242
   scheduling_policies 242
client-propagated priority model 14, 266
communication resources 213
configuring 263
   ORB 263
corba_messaging build flag 262
create_mutex() operation 223
create_named_mutex() operation 259
create_priority_banded_connection_policy() operation 228
create_reference_with_id_and_priority() operation 271
create_reference_with_priority() operation 271
destroy_mutex() operation 223
differentiated services (diffserv) 252
   codepoints (DSCP) 252
   diffserv field (DF) 252
distributable thread 213, 217, 235–238, 240
dynamic scheduling 14, 211, 213, 235, 260
dynamic threads 224
earliest deadline first (EDF) scheduling 235, 243
EDF_Scheduling module 243
enable_network_priority attribute 231, 252
enabling support in TAO 262
end-to-end priority propagation 213, 227
example 218, 270, 278
explicit binding 213, 228
extensions 14, 215
features 234
fixed priority scheduling 235, 242
FP_Scheduling module 243
FP_Scheduling::FP_Scheduler interface 243
FP_Scheduling::SegmentSchedulingParameterPolicy interface 243
general inter-ORB protocol (GIOP) 231
implementation 246
INCOMPATIBLE_SCHEDULING_DISCIPLINES exception 242
least laxity first (LLF) scheduling 244
LLF_Scheduling module 244
lock() operation 222
Max_Utility_Scheduling module 245
maximize accrued utility (MAU) scheduling 245
maxPriority 220
MIF_Scheduling module 246
minimum_corba build flag 262
minPriority 220
modules 216
most important first (MIF) scheduling 246
mutex
   named 259
Mutex interface 240
MutexNotFound exception 259
named mutexes 259
NetworkPriorityMapping interface 256–258
NetworkPriorityMappingManager interface 257–259
object binding 228
open_named_mutex() operation 259
options
   ORBSchedPolicy 263–264
   ORBScopePolicy 263, 265
ORB 217, 221, 256, 263
   configuration 263
ORBNetworkPriorityMapping interface 257
ORBNetworkPriorityMapping option 264
ORBPriorityMapping option 251, 263–264
ORBPriorityRange option 217
ORBSchedPolicy option 264
ORBScopePolicy option 265
overview 212
policies 214
portable interceptors 235, 240, 260–261
portable object adapter (POA) 215, 218
portable priorities 14, 213
priority 222
priority mapping 215, 220, 222, 247, 250
   continuous 247, 264
   direct 247–248, 263–264
   linear 247, 249, 263
   maxPriority 220
minPriority 220
network 252–255, 257–258
NetworkPriorityMapping interface 256–258
NetworkPriorityMappingManager interface 257–259
ORBPriorityMapping option 251, 263–264
ORBPriorityRange option 217
PriorityMapping interface 221, 251
PriorityMappingManager interface 251
priority models 269, 272
  client-propagated 266, 268
  server-declared 269, 272
priority propagation 227–228
priority-banded connections 228
PriorityBands 229
PriorityMapping interface 221, 251
PriorityMappingManager interface 251
PriorityModelPolicy policy 228, 268
private connections 230
PrivateConnectionPolicy policy 230
PROCESS policy 265
processor resources 213
protocol properties 213, 230–231
ProtocolList 232
quality of service (QoS) 213
  policy framework 215
rate-monotonic scheduling
    See real-time CORBA, fixed priority scheduling
real-time operating system (RTOS) 220
reliable oneway invocation 234
request buffering 214
resources 213
rt_corba build flag 262
RT_ORB_Loader 251, 263
RTCORBA module 216, 474, 561, 565
RTCORBA.pidl 217
RTCORBA::Current interface 157, 215, 221, 237
RTCORBA::Mutex interface 215, 222
RTCORBA::PriorityMapping interface 222
RTCORBA::RTORB interface 215
RTPortableServer extension 10, 215–219, 262, 271
RTPortableServer.pidl 217
RTScheduler.pidl 217, 240
RTScheduling extension 217
RTScheduling::Current interface 217, 235–238, 260–261, 273
RTScheduling::DistributableThread interface 235
RTScheduling::ResourceManager::lock() operation 240
RTScheduling::ResourceManager::try_lock() operation 240
RTScheduling::ResourceManager::unlock() operation 240
RTScheduling::Scheduler interface 240, 260
RTScheduling::ThreadAction interface 236, 273, 275
SCHED_FIFO policy 264
SCHED_OTHER policy 264
SCHED_RR policy 264
schedulers 213
scheduling 234
  ResourceManager 240
scheduling segment 236–240, 243, 260, 273, 276
scope_policy 265
SERVER_DECLARED policy 269
server-declared priority model 14, 269
ServiceContexts 228
shared memory inter-ORB protocol (SHMIOP) 231
standard synchronizers 213
static scheduling 211, 213, 235
static threads 224
support 262
SYSTEM policy 265
TAO_HAS_CORBA_MESSAGING macro 262
TAO_HAS_MINIMUM_CORBA macro 262
TAO_HAS_RT_CORBA macro 262, 1385
TAO_Local_RefCounted_Object class 251, 257
TCPProtocolProperties 231, 252
thread action objects 236
thread borrowing 224
thread lanes 224
thread pools 15, 213, 223, 225
ThreadpoolId 215, 224
ThreadpoolLane 215
ThreadpoolLanes 225
ThreadpoolPolicy 224
timeouts 234
to_CORBA () operation 221, 248, 256
to_native () operation 221, 248
to_network () operation 256
transmission control protocol/internet protocol (TCP/IP) 231
try_lock() operation 223
UNIX inter-ORB protocol (UIOP) 231
unlock() operation 223
utility function 245

real-time event service (RTES) xxx, xxxvi, 14, 825, 871–877, 879, 883, 886, 892, 894–895, 904–940, 1465
architecture 872
attributes 900
  consumer_poa 900
  consumer_reconnect 900
  disconnect_callbacks 900
  scheduler 900
  supplier_poa 900
  supplier_reconnect 900
bit-mask filters 889
connecting 876, 879, 881
constructing filters 886
consumer proxy 872–873, 878–879, 881, 993
CORBA gateway 905
deadline timeouts 895
disconnecting 881
dispatching model 873
event
correlation 873, 891
data 884
dispatching 873
filtering 873
header 883, 905
structure 883
type 877, 879, 896
event channel
dispatching strategy 847
resource factory 915
options 921
  ECCConsumerControl 918, 922, 943–948
  ECCConsumerControlPeriod 918, 923
  ECDisplaying 924
  ECDisplayingThreads 926
  ECFFiltering 927
  ECOObserver 928
  ECProxyConsumerLock 929
ECProxyPushConsumerCollection 930
ECProxyPushSupplierCollection 932
ECProxySupplierLock 934
ECScheduling 935
ECSupplierControl 918, 936
ECSupplierControlPeriod 918, 937
ECSupplierFiltering 938
ECTimeout 939
ECUseORBId 940

servants 898
event channel attributes 900
event structure 883
customizing 885
options
  TAO_LACKS_EVENT_CHANNEL_ANY 885
  TAO_LACKS_EVENT_CHANNEL_OCTET_SEQUENCE 885
  TAO_LACKS_EVENT_CHANNEL_TIMESTAMPS 885

Event_Service program
  command line options 914
EventData 884
EventHeader 883
EventType special values
  ACE_ES_EVENT_ANY 884
  ACE_ES_EVENT_DEADLINE_TIMEOUT 884
  ACE_ES_EVENT_INTERVAL_TIMEOUT 884
  ACE_ES_EVENT_SHUTDOWN 884
  ACE_ES_EVENT_UNDEFINED 884

example 874
  local event channel 898
feature control options 917
  ECFiltering 917
  ECObservable 917
  ECProxySupplierLock 917
  ECScheduling 917
  ECSupplierFilter 917
  ECSupplierFiltering 917
federating event channels 902, 913
  IP multicast 911
  UDP 907
filtering
  and correlation 886
  building 887
combinations 888
complex filters 886
conjunction groups 890
construction 894
custom filters 894
disjunction groups 889
event type 887
logical AND groups 891
negating the logic of filters 891
nesting groups 892
null filters 894
rejection filters using bit masks 893
source ID 888
interval timeouts 895
locking options 918
   ECPProxyConsumerLock 918
   ECPProxySupplierLock 918
managing connections 880
observers 901
old 784
operations
   resume_connection() 882
   suspend_connection() 882
ORB service libraries 784
prefix filter builder 887
priority timers 873
proxy collection options 919
push consumer interface 878
reconnecting
   consumers 882
   suppliers 882
rejection filters 893
resource factory
   consumer and supplier control options 918
resuming consumer connections 882
specifying and constructing filters 886
subscription and filtering 873
supplier proxies 873, 881
suspending consumer connections 882
timeout 895
events 873

real-time scheduling 234, 273–278
real-time scheduling service
  ORB service libraries 784
rebind() operation 366, 845
receive_exception() operation 283
receive_other() operation 283
receive_reply() operation 283
receive_request() operation 261, 284
receive_request_service_context() operation 284
receptacle
  CORBA Component Model (CCM) 1215–1216, 1219, 1265, 1273–1277, 1281–1283, 1307, 1317, 1319
reclaim_reactor() operation 692
reconfigure() operation 602, 616
record protocol 1094
recurse option 38, 44
recv() operation 509, 524
red-black tree 850
reference counting 363
reference_to_servant() operation 1195
register() operation 1189
register_handler() operation 509
register_new_factory() operation 324
register_proxy_factory() operation 336–337
register_value_factory() operation 319
registering interceptors 286
  IOR 307
regular value types 330
rejection filters using bit masks 893
RejectNewEvents property 961, 963–964, 966–967
relationship between ACE and TAO 15
relative option 39
RELATIVE_RT_TIMEOUT_POLICY_TYPE 169, 172
release() operation 363, 541
reliable oneway invocation 175, 234
  SyncScope policy 175
remove directive 602
reply handler 147, 150
  exception 155
  non-exception 151
  operations 147
repo build flag 1378, 1381
RepositoryId 285–286, 1050–1051
RepositoryManager program 1343, 1354, 1356

request
  buffering 214
  demultiplexing
    active 10, 15
  interceptors 282
  invocation 530
  processing 531

request invocation 530–531
request processing 531–554

RequestProcessingPolicy policy 359

requires_explicit_endpoint() operation 497

resolve() operation 828
resolve_str() operation 808

  ACE_Data_Block 693
  ACE_Message_Block 693
  ACE_QtReactor 692
  ACE_QtReactor class 708
  ACE_Reactor 692
  ACE_XtReactor 692
  advanced 380–381, 690, 692, 694, 697, 705, 710–712, 734
  advanced resource factory options 710, 730
  Advanced_Resource_Factory configuration 380, 558–559, 569–570
  allocator configuration 704
  amh_response_handler_allocator() operation 704
  ami_response_handler_allocator() operation 704
  byte-order flag 379, 725
  cache management strategies 695
  cache_maximum() operation 696–697
  CDR conversion allocators 689, 693
  Client_Strategy_Factory configuration 595–596, 763, 770–772, 774
  codeset identifiers and translators 703
  common data representation (CDR) 693
  connection cache management strategies 695
  create_cached_connection_lock() operation 696–697
  create_corba_object_lock() operation 705
  create_flushing_strategy() operation 695
  create_lf_strategy() operation 692
create_purging_strategy() operation 696–697
custom factory 725
custom interoperable object reference (IOR) parsers 699
default 381, 690, 692, 694–695, 697–698, 705, 711
event channel 846–852, 899, 915–919
flushing strategy 695
general inter-ORB protocol (GIOP) 693, 725
get_parser_names() operation 700
get_protocol_factories() operation 699
init_protocol_factories() operation 699
input_cdr_allocator_type_locked() operation 694
input_cdr_buffer_allocator() operation 693, 705
input_cdr_dblock_allocator() operation 693, 705
input_cdr_msgblock_allocator() operation 693, 705
interface definition 690
internet inter-ORB protocol (IIOP) 692
lite protocols 726
locked_transport_cache() operation 696–697
magic number 379, 725
max_muxed_connections() operation 696–697
options 713
notify manager 1022–1023
ORBAMHResponseHandlerAllocator 705, 731
ORBAMIResponseHandlerAllocator 705, 732
ORBCCharCodesetTranslator 704
ORBConnectionCacheLock 694, 698, 715, 734
ORBConnectionCacheMax 698, 716
ORBConnectionCachePurgePercentage 698, 717, 733
ORBConnectionPurgingStrategy 697, 712, 733
ORBCorbaObjectLock 706
ORBFlushingStrategy 576, 580, 695, 719
ORBInputCDRAlocator 558–559, 694–695, 712, 715, 734
ORBiorParser 702–703, 720
ORBmuxedConnectionMax 698, 721
ORBNativeCharCodeset 704
ORBNativeWCharCodeset 704
ORBOBJECTKeyTableLock 699, 724
ORBPriorityMapping 263
ORBReactorMaskSignals 727
ORBReactorThreadQueue 712, 735–736
ORBReactorType 557–559, 568–570, 683, 693, 712, 737–738
fl 737
msg_wfmo 737
select_mt 737
select_st 737
tk_reactor 737
tp 737
wfmo 737
ORBResourceUsage 707, 728
ORBSchedPolicy 264
ORBWCharCodesetTranslator 704
output_cdr_buffer_allocator() operation 694
output_cdr_dblock_allocator() operation 694
output_cdr_msgblock_allocator() operation 694
protocol factory 376, 390, 689, 699, 726, 1116–1117, 1120, 1122–1123
purge_percentage() operation 696–697
purging strategies 695
  first in first out 696
  least frequently used 696
  least recently used 696–697
  no operation (NOOP) 696
Qt GUI toolkit 690, 707, 711
reactor management 689, 692
reclaim_reactor() operation 692
resource usage strategy 706, 728
  eager 691, 706–707
  lazy 691, 706–707
resource_usage_strategy() operation 706–707
select_st option 692
TAO_CONNECTION_CACHE_MAXIMUM macro 716
TAO_Flushing_Strategy class 695
TAO_LF_Strategy class 692
TAO_QtResource_Factory class 708
TAO_Resource_Factory class 690–691
TAO_Strategies library 711
TAO_Transport_Cache_Manager class 697
TAO_XT_Resource_Factory class 709
UNIX inter-ORB protocol (UIOP) 692, 726
use_locked_data_blocks() operation 694
WaitForMultipleObjects() operation 737
X windows toolkit 707, 709
Xt 690, 711
XtAppContext 709

**resource usage strategy** 706, 728
  eager 691, 706–707
  lazy 691, 706–707


**resource_usage_strategy() operation** 706–707

**resume directive** 603

**resume_connection() operation** 993

**resume_handler() operation** 470, 472

**RETAIN policy** 356, 477, 1380

**Riehle, Dirk** 1466

**Rohnert, Hans** 16, 404, 563, 582, 1467

**root naming context** 828

**RT CORBA**

See real-time CORBA

**RT_Collocation_Resolver class** 265, 640

**rt_corba build flag** 262, 1378, 1385

**RT_ORB_Loader** 251

  options
    ORBNetworkPriorityMapping 264
    ORBPriorityMapping 263
    ORBSchedPolicy 263–264
    ORBScopePolicy 263, 265

**RT ORB Loader, configuration** 257, 263

**RTCORBA module** 64, 135, 211–234, 246, 250–259, 262–272, 474, 561, 565

**RTCORBA::Mutex interface** 222, 240

**RTCORBA::Priority interface** 222

**RTPortableServer extension** 10, 215–219, 262, 271

**RTScheduling extension** 216–217

**RTScheduling::Current interface** 238

**RTScheduling::ResourceManager interface** 240, 242

**RTScheduling::Scheduler interface** 240–241, 260

**rtti build flag** 1377, 1380

**Ruh, William** 372, 1467

**run() operation** 537, 550, 582, 1165

**rwho build flag** 1378, 1386
sample

**SCHED_FIFO policy** 223, 263–264

**SCHED_OTHER policy** 264

**SCHED_RR policy** 223, 264

schedulers 213

scheduling
real-time 273–278

scheduling segment 236–240, 243, 260, 273, 276

scheduling service 251, 875, 877, 901, 914, 917, 927, 935

**scheduling_discipline_name attribute** 242

**scheduling_policies attribute** 242


SCIOP

*See SCTP inter-ORB protocol (SCIOP)*

**scope_policy policy** 265

SCTP

*See SCTP inter-ORB protocol (SCIOP)*

*See stream control transmission protocol (SCTP)*

**scpt build flag** 1379, 1387

**SCTP inter-ORB protocol (SCIOP)** xxviii, 11, 374, 396

SECIOP

*See secure inter-ORB protocol (SECIOP)*

**secret-key cryptography** 1088

**secure inter-ORB protocol (SECIOP)** 1075, 1078–1079

interoperability 1075, 1078–1079

**secure socket layer inter-ORB protocol (SSLIOP)** 11, 374, 388, 1065, 1086–1087, 1093, 1106, 1108, 1111, 1114–1118, 1120, 1122–1135, 1445

endpoints 649

factory 388

options 1132

SecurityLevel1 module 1124

SSLAcceptTimeout option 1114

SSLAuthenticate option 1115, 1134

SSLCertificate option 1114, 1133

SSLDHparams option 1114

SSLIOP::Current interface 1125
secure sockets layer (SSL) protocol 11, 374, 388, 649, 1067, 1069, 1079, 1087–1098
   architecture 1092
   example 1097
   SSLAuthenticate option 1134
   SSLCertificate option 1133

security 1065, 1069
   access identity 1070
   administration 1073
   alert protocol 1097
   application
      control 1073
      enforcement 1073
   architecture 1079, 1081, 1085
   asymmetric encryption 1089
   audit identity 1070
   auditing 1072
   binding 1091
      client and target 1084
   building 1106
      on UNIX 1107
      on Windows 1108
   security-aware applications 1110
   certificate 1091, 1099
      authority 1091, 1100
      commands summary 1106
      issuing 1103
      multiple authorities 1112
      request 1101, 1103
   change cipher specification protocol 1096
   ciphertext 1088
   common secure interoperability (CSI) packages 1066, 1075, 1078–1079
      CSI 0 1075, 1078
      CSI 1 1066, 1075, 1078
      CSI 2 1066, 1075, 1078
   context information 1084
   controlling
      message protection 1118
      peer authentication 1119
creating
  certificate authority 1100
  certificate requests 1101
credential 1073
CSI ECMA protocol 1075, 1079
DCE Common Inter-ORP protocol (DCE-CIOP) 1075, 1078
delegation of attributes 1071
digital signatures 1090
distinguished name 1091
environment setup 1079, 1099, 1111
establish trust policy 1118
example 1097, 1115
  building libraries 1120
  security policy enforcement 1126
  SSL session 1097
feature packages 1074
  common secure interoperability (CSI) 1078
  main functionality 1076
  optional 1077
  SECIOP + DCE-CIOP interoperability 1078
  SECIOP interoperability 1078
  security mechanism 1078
    CSI ECMA protocol 1079
    generic security service (GSS) protocol 1079
    secure sockets layer (SSL) protocol 1079
    simple public key mechanism (SPKM) protocol 1078
  security replaceability 1077
    ORB services 1077
    security services 1077
generic security service (GSS) 1079
handshake protocol 1093
identity
  attributes 1071
  binding 1091
  certificates 1091
implementation architecture 1079
initiating principal 1071
Kerberos 1075, 1079, 1089
key
  pair 1089
  removing pass phrases 1105
  server 1089
Level 1  1076
Level 2  1076
libraries
  building  1120
  testing  1109
message
  authentication codes  1090
  confidentiality  1087
  controlling protection  1118
  integrity  1090
  protection  1118
messaging  1073
non-repudiation  1072, 1074, 1077
OpenSSL  374, 1068, 1088, 1096, 1099–1101, 1103, 1105–1108, 1110–1111, 1125
optional packages  1077
options
  SSLAuthenticate  1134
  SSLCertificate  1133
  SSLNoProtection  1132–1133
  SSLPrivateKey  1132, 1134
originating principal  1071
peer authentication  1119
  controlling  1119
plaintext  1088
policy
  controlling  1065, 1068, 1073, 1118, 1445
  example  1120
  enforcing  1065, 1073, 1124, 1445
  example  1126
  establish trust  1118
principal  1070
  identification and authentication  1070
privacy enhanced mail (PEM)  1099
private key  1090
privilege attributes  1071
public key  1090
public-key cryptography  1089–1090
quality of protection (QoP)  1118
record protocol  1094
reference model  1070
removing key pass phrases  1105
secret-key cryptography 1088
secure inter-ORB protocol (SECIOP) 1075, 1078–1079
secure socket layer inter-ORB protocol (SSLIP) 1065, 1114, 1445
secure sockets layer (SSL) protocol 11, 374, 388, 649, 1067, 1069, 1079, 1087–1098
  architecture 1092
Security module 1118–1119
SecurityAdmin module 1074, 1076
SecurityLevel1 module 1066, 1074, 1076, 1087, 1124–1125
  Current interface 1124–1125
SecurityLevel2 module 1066, 1074, 1076, 1087, 1125
SecurityLevel3 module 1066, 1125
self-signed certificate 1092
signing certificate requests 1103
simple delegation 1072
symmetric encryption 1088
ticket 1089
transparent protection 1072
transport layer security (TLS) 1087
unaware 1065, 1068, 1445
  application 1110
  example 1115
  building executables 1109
Security module 1076, 1118–1124, 1394, 1447
security service 13
SecurityAdmin module 1074, 1076
SecurityLevel1 module 1066, 1074, 1076, 1087, 1124–1125
SecurityLevel2 module 1066, 1074, 1076, 1087, 1125
SecurityLevel3 module 1066, 1125
select reactor
  single-threaded 568
select() operation 403, 569–570, 737
select_mt option 569–570, 712, 737
select_st option 557–559, 568–569, 692, 712, 737
send() operation 504
send_exception() operation 284
send_message() operation 506
send_message_shared() operation 507, 524
send_message_shared_i() operation 512, 523
send_other() operation 285
send_poll() operation 283
send_reply() operation 284
send_request() operation 283, 505
send_synchronous_message_i() operation 516
sendc_prefix 141, 143, 158
SEQUENCE_EVENT type 984, 990, 1009, 1013
SequenceProxyPushConsumer interface 951, 1009
SequenceProxyPushSupplier interface 1013–1014
SequencePushConsumer interface 954, 1005, 1011, 1014
SequencePushSupplier interface 954, 1005–1006, 1009
servant 9, 160, 162
    activation
    USE_SERVANT_MANAGER 359
    locator 356
    manager 356, 629
    NON_RETAIN 359
        RequestProcessingPolicy 359
        ServantRetentionPolicy 359
ServantLocator interface 356–357, 359
servant-per-AMI-call strategy 156–157
ServantRetentionPolicy policy 356, 359
server concurrency 741
server interceptors 304
    interception points 284
    request 284
server role 529, 531
server strategy factory 134, 556, 558, 560–562, 593, 595, 620, 739–742, 744,
    746–748, 759–762
    active object map 747
    default options 748
    demultiplexing strategies 743
    interface definition 740
    options
        ORBAActiveHintInIDs 747, 749
        ORBAActiveHintInPOANames 746, 750
        ORBAActiveObjectMapSize 747, 751
        ORBConcurrency 556, 558, 560, 562, 742, 753
        ORBPOALock 558, 742, 755
        ORBPOAMapSize 746, 756
        ORBSSystemIDPolicyDemuxStrategy 748, 757
        ORBThreadFlags 743, 758
        ORBThreadPerConnectionTimeout 561–562, 743, 759
        ORBUniqueIDPolicyReverseDemuxStrategy 748, 761
        ORBUserIDPolicyDemuxStrategy 748, 762
POA map 746
Server_Strategy_Factory configuration 556, 558, 560, 562, 595–596, 739, 749–762

SERVER_DECLARED policy 228, 269–272
server-differentiated-reply strategy 156–157
ServerRequestInfo interface 285

service composition
CORBA Component Model (CCM) 1229

service configurator 11, 591–592, 594–606, 608–616, 621, 701, 710–711, 713, 852
configuring TAO clients and servers 592
control options 597
creation 609
directives 595–596, 602
DTD 604
initialization and finalization 608
interface definition 607
manager 615
objects 606
specializing factories 594
state 614
static service example 613
csv.conf file 381, 390, 562, 594, 599, 601, 682, 915, 941, 1120, 1123
See also ACE, service configurator

service configurator file 558

service manager 601–603
ServiceContexts 228

services 781–786
audio/video streaming 13, 783
concurrency control 4, 13, 784
Cos prefix 782
customizing access 782
data distribution 13, 1445
event xxx, xxxvi, 13, 784, 825–827, 871–879, 886, 952, 959
implementation repository 366, 1022, 1167
ImR_Activator 1171
interface repository xxx, 13, 104, 1049–1052, 1054–1056, 1062–1063
life cycle 13, 784
load balancing 13, 334, 642, 764, 784
log 13
NamingViewer utility 815
nsadd utility 814
nsdel utility 814
nslist utility 814
ORB service libraries 334, 784
overview of TAO 781
property 13
real-time event service (RTES) xxx, xxxvi, 14, 825, 871–877, 879, 883, 886, 892, 894–895, 904
scheduling 251, 875, 877, 901, 914, 917, 927, 935
security 13, 784
selectively building TAO services 1398
TAO ORB libraries 783
time 13, 172, 244
trading 13, 624, 633–634, 636–638, 655, 684, 781, 836

services keyword
CORBA Component Model (CCM) 1231

session composition
CORBA Component Model (CCM) 1228

session keyword
CORBA Component Model (CCM) 1231

set_locator() operation 368–369
set_policy_overrides() operation 170, 355, 1076, 1083, 1087
set_qos() operation 964
set_scheduling_parameter() operation 242
set_session_context() operation 1253–1254, 1259, 1268, 1270–1271, 1279
set_slot() operation 311

shared memory inter-ORB protocol (SHMIOP) 11, 231, 374, 382–385, 393, 399, 403, 415–417, 432, 482, 494, 497, 634, 644, 648, 651, 655–657, 692, 725, 767, 808

endpoint 648
text 651
factory 383
SHMIOP_Factory configuration 384, 725

shared_libs build flag 1378, 1382–1383
shared_libs_only build flag 1378, 1383

SHMIOP
See shared memory inter-ORB protocol (SHMIOP)

shutdown() operation 534, 541–546, 1128–1129, 1131, 1139, 1164
shutting down 543
SII
See static invocation interface (SII)

simple TCP transport
data-centric publish-subscribe (DCPS) 1181, 1183, 1187–1188, 1191, 1198, 1201–1202, 1209

simple UDP transport
data-centric publish-subscribe (DCPS) 1198, 1201–1202

simplex receptacle
CORBA Component Model (CCM) 1275–1276, 1297, 1299, 1306–1307

single thread model 548
SINGLE_THREAD_MODEL policy 547, 551–553

sle build type 36

smart proxies xxx, 6, 92, 331–332, 334–335, 340–342, 345, 350–352
Unchecked_narrow() operation 339
asynchronous method invocation (AMI) 332
base_proxy_variable 338
batch processing 334
choosing between target objects 334
client-side caching 333
creating 339
default proxy factory 337, 339
example 344–351
  logging 344
executing a sequence of operations 334
factory object 342
framework 334
get_proxy() operation 337–338
IDL compiler 332
instance() operation 338
logical target object 334
oneway 332
overridden member functions 338
proxy factory adapter 336, 339
register_proxy_factory() operation 336–337
smart proxy base class 337
smart proxy class 332, 341, 348
smart proxy factory class 339, 342, 346
smart_proxies build flag 335, 341
tao/SmartProxies/Smart_Proxies.h 335, 341
TAO_Default_Proxy_Factory class 335–336
TAO_Proxy_Factory_Adapter class 335
TAO_Smart_Proxy_Base class 335–338, 348
unregister_proxy_factory() operation 336, 343
use cases 333
writing and using 340

smart_proxies build flag 335, 341
SO_LINGER socket option 660
sockets
   CLOSE_WAIT state 584
spawn() operation 235, 237–238, 273
specializing factories 594
split build flag 1379, 1387
SSI
   See static skeleton interface (SSI)
SSL
   See secure sockets layer (SSL) protocol
ssl build flag 1107–1108, 1378, 1386
SSL_CERT_DIR environment variable 1111–1112
SSL_CERT_FILE environment variable 1111–1112
SSL_EGD_FILE environment variable 1111
ssl_port option 645
SSL_RANK_FILE environment variable 1111
SSL_ROOT environment variable 1107–1108
SSLIOP
   See secure socket layer inter-ORB protocol (SSLIOP)
Stal, Michael 16, 404, 563, 582, 1467
standard synchronizers 213
state member
   value type 317
static
   loading 490
   threads 224
static invocation interface (SII) 9, 12
static option 39
static scheduling 211, 213, 235
static skeleton interface (SSI) 9, 12
static_libs build flag 1378, 1382–1383
static_libs_only build flag 1378, 1382–1383
static_link build flag 1379, 1388
Stevens, W. Richard 1468
STLport 1378, 1386
stlport build flag 1378–1379, 1386
strategies 593
activation-per-AMI-call 156–157
blocking 719
connect 770
first in first out (FIFO) 696
leader/followers 695, 719
leader_follower 576
least frequently used (LFU) 696, 712, 733
least recently used (LRU) 696–697, 712, 733
no operation (NOOP) 696
reactive 719
servant-per-AMI-call 156–157
server-differentiated-reply 156–157

strategy acceptor template 422
strategy connector template 446
strategy pattern 15
stream control transmission protocol (SCTP) 11, 396
string_dup() operation 27
structured event types 825
STRUCTURED_EVENT type 984, 990
StructuredProxyPushConsumer interface 982, 984, 986
StructuredProxyPushSupplier interface 951, 990, 1005
StructuredPushSupplier interface 953, 979, 985
stubs and skeletons 9, 23, 83
subscriber
CORBA Component Model (CCM) 1224
subscription and filtering
real-time event service (RTES) 873
subscription_change() operation 957–958, 979–980, 991, 996–997, 999–1000, 1006, 1014
supplier proxies 873
real-time event service (RTES) 873
SupplierAdmin interface 950, 957, 983–984, 994, 1001–1002, 1008
support
commercial xxxiv
Object Computing, Inc. (OCI) xxxiv
supports keyword
CORBA Component Model (CCM) 1298, 1370
suspend directive 603
suspend_connection() operation 993
suspending and resuming consumer connections
real-time event service (RTES) 882
svc() operation 276, 508, 567, 811–812, 1254, 1256–1257, 1259, 1275, 1278–1281
svc.conf file 384–385, 390, 594, 599, 601, 1120, 1122
SYNC_NONE 175–176, 179
SYNC_WITH_TARGET 176
SYNC_WITH_TRANSPORT 176
synchronizers
standard 213
SyncScope policy 167, 175–177, 179
SYNC_NONE 175–176, 179
SYNC_WITH_SERVER 175
SYNC_WITH_TARGET 176
SYNC_WITH_TRANSPORT 175–176
system requirements 18
See also building
Syyid, Umar 1466

T

tagged components 305, 307, 681
TaggedComponent() operation 309
take() operation 1196
take_instance() operation 1196
take_next_instance() operation 1196
take_next_sample() operation 1196
TAO xxv, 3, 691
architecture 8
catio utility 309
contributors 1449–1463
design goals 4
development history 6
high performance 14
minor codes 131
ORB service libraries 334
real-time CORBA support 14, 262
real-time event service (RTES) 14
relationship to ACE 15
source code distribution 1405, 1413
supported platforms xxxiii
system requirements 18
TAO_ROOT environment variable 86, 88
version xxxii

**TAO classes**

RT_Collocation_Resolver 265, 640

TAO_Accept_Strategy 419, 421, 425


TAO_Acceptor_Registry 134, 408, 691

TAO_Base_Transport_Property 472

TAO_CEC_EventChannel 827, 834

TAO_CEC_EventChannel_Attributes 835

TAO_CEC_Factory

  interface definition 847

TAO_Connect_Strategy 442, 764, 768

TAO_Connection_Handler 415, 417, 464, 466–468

TAO_Connection_Purging_Strategy 691, 696–697


TAO_Connector_Registry 133, 135, 407, 691

TAO_Creation_Strategy 419, 421, 424

TAO_Default_Proxy_Factory 335–336

TAO_Default_Resource_Factory 418

TAO_EC_Event_Channel_Attributes 900–901

TAO_EC_Factory

  dispatching 916


TAO_Flushing_Strategy 513, 516, 691, 695

TAO_GIOP_Synch_Invocation 518

TAO_IID_Acceptor 408

TAO_IID_Profile 409, 481, 483

TAO_IID_Profile_Factory 407, 490

TAO_IOR_Parser 700, 720

TAO_LF_Strategy 692

TAO_Local_RefCounted_Object 251, 257, 306, 353, 363, 1239, 1251

TAO_MProfile 133, 419–420, 433–436, 442, 477–479

TAO_Naming_Client 810–811

TAO_Naming_Server 811–812

TAO_Notify_EventChannelFactory_i 1016

TAO_Priority_Mapping_Manager  251
TAO_ProtocolFactorySet  691
TAO_Proxy_Factory_Adapter  335
TAO_QtResource_Factory  708–709
TAO_Resource_Factory  690–691
TAO_Server_Strategy_Factory  426, 452, 740–741
TAO_Smart_Proxy_Base  335–338, 348
TAO_Transport_Cache_Manager  472, 499, 502, 697
TAO_UIOP_Acceptor  419, 427–430, 432–436, 438–440, 495
TAO_UIOP_Profile  483
TAO_UIOP_Properties  420, 468
TAO_UIOP_Protocol  487–489, 494–497, 506
TAO_UIOP_Transport  468, 504, 507, 509
TAO_Unknown_Profile  461
TAO_Wait_Strategy  499, 502, 764, 766
TAO_XT_Resource_Factory  709–710

**TAO configuration**  xxviii, xxx, 5, 591
client strategy factory configuration  573–574, 580–581, 583, 585, 595–596, 763, 770–772, 774
directives  594
dynamic  376, 559, 596, 608, 612
dynamic components  596
resume  603, 614
static  559, 595, 608, 690, 739, 763, 847, 915, 941
suspend  603, 614
event channel servants
real-time event service (RTES)  898
object factory  592–593
server strategy factory configuration  556, 558, 560, 562, 563, 593, 595–596, 739, 749–762
source code
on UNIX  1393
specializing factories  594
static directive  595
### TAO libraries

- ORB service 642, 764  
- TAO_BiDirGIOP 182  
- TAO_CosConcurrency 784  
- TAO_CosEvent 784  
- TAO_CosLifeCycle 784  
- TAO_CosLoadBalancing 784  
- TAO_CosNaming 98, 784, 792  
- TAO_CosNaming_Serv 792  
- TAO_CosNaming_Skel 792  
- TAO_CosNotification 784  
- TAO_CosProperty 784  
- TAO_CosTime 784  
- TAO_CosTrading 784  
- TAO_DsEventLogAdmin 784  
- TAO_DsLogAdmin 784  
- TAO_DsNotifyLogAdmin 784  
- TAO_Messaging 167, 180  
- TAO_PortableGroup 375, 390  
- TAO_RTCORBA 135, 251, 257, 262  
- TAO_RTEvent 784, 885  
- TAO_RTEventLogAdmin 784  
- TAO_RTOLDEvent 784  
- TAO_RTPortableServer 262  
- TAO_RTSched 784  
- TAO_RTSchedEvent 784  
- TAO_Security 784  
- TAO_SmartProxies 341, 343  
- TAO_Svc_Utils 785

### TAO macros

- TAO_CONNECTION_CACHE_MAXIMUM 716  
- TAO_CONNECTION_REGISTRY_NO_USABLE_PROTOCOL 133  
- TAO_DEFAULT_IMPLREPO_SERVER_REQUEST_PORT 653  
- TAO_DEFAULT_NAME_SERVER_REQUEST_PORT 665, 788  
- TAO_DEFAULT_SERVER_ACTIVE_OBJECT_MAP_SIZE 743, 751  
- TAO_DEFAULT_SERVER_POA_MAP_SIZE 756  
- TAO_DEFAULT_TRADING_SERVER_REQUEST_PORT 684  
- TAO_HAS_AMI 1385  
- TAO_HAS_AMI_CALLBACK 1385  
- TAO_HAS_AMI_POLLER 1385
TAO_HAS_CORBA_MESSAGING 262, 1385
TAO_HAS_INTERCEPTORS 1385
TAO_HAS_MINIMUM_CORBA 262, 1383, 1393, 1398, 1440
TAO_HAS_MINIMUM_POA 1398
TAO_HAS_RT_CORBA 262, 1385
TAO_INVOCATION_LOCATION_FORWARD_MINOR_CODE 132
TAO_INVOCATION_RECV_REQUEST_MINOR_CODE 133
TAO_INVOCATION_SEND_REQUEST_MINOR_CODE 132
TAO_MPROFILE_CREATION_ERROR 133
TAO_POA_DISCARDING 132
TAO_POA_HOLDING 132
TAO_RESUME_CONNECTION_HANDLER 472
TAO_STD_PROFILE_COMPONENTS 681
TAO_TAG_UIOP_PROFILE 427, 454
TAO_THREAD_PER_CONNECTION_TIMEOUT 759
TAO_TIMEOUT_CONNECT_MINOR_CODE 133
TAO_TIMEOUT_RECV_MINOR_CODE 134
TAO_TIMEOUT_SEND_MINOR_CODE 133–134
TAO_UNHANDLED_SERVER_CXX_EXCEPTION 133
TAO_USE_LAZY_RESOURCE_USAGE_STRATEGY 707, 728

TAO_Acceptor_Registry class 691
TAO_BiDirGIOP library 182
TAO_CEC_EventChannel class 827
TAO_Connection_Purging_Strategy class 691, 697
TAO_Connector_Registry class 691
TAO_DEFAULT_SERVER_ACTIVE_OBJECT_MAP_SIZE macro 751
TAO_DEFAULT_SERVER_POA_MAP_SIZE macro 756
TAO_DEFAULT_THREAD_PER_CONNECTION_TIMEOUT macro 743
TAO_EC_Event_Channel_Attributes class 900–901
TAO_Flushing_Strategy class 691, 695
TAO_HAS_MINIMUM_CORBA macro 1398
TAO_HAS_MINIMUM_POA macro 1398
tao_idl program 23–24, 53, 83–91, 95, 97–98, 162, 361, 1368
TAO_IDL_PREPROCESSOR environment variable 86, 88
TAO_IDL_PREPROCESSOR_ARGS environment variable 88
tao_idl3_to_idl2 program 1365–1366, 1368–1369
tao_ifr program 1051, 1053–1054, 1062–1063
options 1053–1054
tao_imr program 1160
TAO_IOR.Parser class 700
TAO_Local_RefCounted_Object class 1239, 1251
TAO_Naming_Client class 810–811
TAO_Naming_Server class 811–812
TAO_Notify_EventChannelFactory_i class 1016
TAO_ORB_Core class 620
TAO_ORB_DEBUG environment variable 632
TAO_ORBENDPOINT environment variable 620, 642, 646
TAO_ProtocolFactorySet class 691
TAO_Resource_Factory class 691
TAO_ROOT environment variable 22, 86, 88
TAO_THREAD_PER_CONNECTION_TIMEOUT macro 759
TAO_Transport_Cache_Manager class 697
TAO_UIOP_Profile class 483
TAO_USE_IMR environment variable 1138–1139, 1144, 1147
TAO_USE_LAZY_Resource_USAGE_STRATEGY macro 707, 728
target_is_a() operation 295
TCP/IP
See transmission control protocol/internet protocol (TCP/IP)
TCP_NODELAY option 669
TCPProtocolProperties 231, 233, 252–255
template files 66
template option 39, 85, 89
templates build flag 1388
testing 1429
building tests 1429
running tests 1430
ACE tests 1430
ORB Services tests 1433
TAO tests 1431
The ACE Programmer’s Guide (APG) book 16, 540, 555, 568, 592, 1466
the_POAManager() operation 26
the_priority attribute 222, 269
THR_BOUND 742, 758
THR_DAEMON 758
THR_DETACHED 742, 758
THR_JOINABLE 566–567
THR_NEW_LWP 566–567, 743, 758
THR_SUSPENDED 758
thread action objects 236
thread borrowing 224
concurrency model 562–565, 570, 586
reactor 555, 562–565, 568–570, 583, 585

**thread-per-connection concurrency model** 387, 392, 426, 471, 554, 559–562, 743, 759, 838

**ThreadPolicy policy** 547, 550–551, 554

**thread-pool concurrency model** 554

**ThreadPool property** 964, 967, 973–975, 1020, 1024

**thread-pool reactor** 225, 737

**ThreadpoolId** 215, 224–226

**ThreadpoolLane** 215

**ThreadpoolLanes** 225

**ThreadpoolLanes property** 965, 967, 973, 975, 1017, 1024

**ThreadpoolPolicy** 224

**threads**
- dynamic 224
- lanes 224
- static 224
- thread specific storage (TSS) 15

**threads build flag** 1378, 1381


**ti build flag** 40–41

**ti option** 39

**time independent invocation (TII)** 139

**time service** 13, 172, 244

**TimeBase module** 172, 244

**Timeout property** 960, 967

**timeouts** 234
- real-time event service (RTES) 895

**TimeT type** 172

**tk_reactor build flag** 712, 737, 1378, 1384

**TLS**
- See transport layer security (TLS)

**to_CORBA() operation** 221, 248, 256, 258

**to_native() operation** 221, 248

**to_network() operation** 256, 258

**to_string() operation** 485, 807

**to_url() operation** 808

**tools**
- Borland Make 29
- GNU Make 28–30
- MakeProjectCreator (MPC) 27
Visual C++ 27, 29–30

topic
  data-centric publish-subscribe (DCPS) 1178–1179, 1183–1184, 1187, 1189–1190, 1192, 1194, 1197–1200, 1203–1205, 1209
top-level package descriptor 1289, 1323–1324, 1326–1327, 1330, 1344, 1356
tp option 737
trading service 13, 624, 633–634, 636–638, 655, 684, 781, 836
  options
    ORBTradingServicePort 624, 684, 788
    TradingServicePort 684, 788
TradingServicePort environment variable 684, 788
training xxxiv, xxxvii
  Advanced CORBA Programming Using TAO xxxvi
  CORBA Programming with C++ xxxv
  CORBA Programming with Java xxxv
  Introduction to CORBA xxxv, 1068
  Introduction to Model-Driven Architecture (MDA) xxxvii
  Introduction to Real-Time Systems xxxvi
  Object Oriented Design Patterns and Frameworks xxxvi
  Using the ACE C++ Framework xxxvi
TRANSIENT exception 103, 107, 132–134, 672, 745–747, 756, 760, 1117
transmission control protocol/internet protocol (TCP/IP) 9, 231, 371–373, 376, 383, 403, 642, 656, 699, 725, 913, 1093, 1437
transport 410, 418
  data-centric publish-subscribe (DCPS) 1181, 1183, 1187, 1191, 1195, 1198, 1201–1202, 1209
  multiplexing strategies 573–574
  pluggable 1181–1182, 1188, 1201
transport factory
  data-centric publish-subscribe (DCPS) 1181, 1188, 1194
transport layer security (TLS) 1087
Trask, Bruce 1466
truncatable keyword
  value types 330
try_lock() operation 223, 240
type codes
  wchar 501, 703–704, 729
type option 39
typed event channel 827
U

UDP
See user datagram protocol (UDP)

UIOP
See UNIX inter-ORB protocol (UIOP)

UIOP_Factory configuration 380–381, 487–488, 725–726
UIOP_Lite_Factory configuration 725

UIPMC
See unreliable IP multicast protocol (UIPMC)

unbind() operation 366

University of California, Irvine
See DÖC Group

UNIX 11, 18, 28, 84, 87, 99, 380, 384, 617, 619, 647–648, 669, 725, 1107, 1332, 1357, 1365, 1369, 1377
building ACE and TAO 1391
configuring source code 1393
customizing ACE and TAO builds 1397
set environment variables 1392
verifying build 1396
See also building

UNIX domain sockets 11

endpoint 648
examples 651
lite 374
UIOP_Factory configuration 380–381, 487–488, 725–726
UIOP_Lite_Factory configuration 725

UNIX Network Programming book 1468
unlock() operation 222–223, 240
unmarshaling 734
unregister_proxy_factory() operation 336, 343
unreliable IP multicast protocol (UIPMC) 374–375, 389–391, 393
untyped event channel 827
update_scheduling_segment() operation 239
use_env option 39
use_locked_data_blocks() operation 694
USE_SERVANT_MANAGER policy 359
uses keyword
  CORBA Component Model (CCM) 1225–1227, 1269, 1299

uses multiple keyword
  CORBA Component Model (CCM) 1226–1227, 1273–1275, 1299

Using the ACE C++ Framework course xxxvi

utilities
  catio 309
  naming service 814
    nsadd 814
    nsdel 814
    nslist 814
  tao_imr 1160

utility function 245

Utility library 1345–1346, 1348–1349, 1352

UTILITY_ROOT environment variable 1349, 1352

validate_connection() operation 355

value box 330

value types 317–330
  abstract keyword 330
  abstract value type 328–330
  class 322
  compliance 330
  DefaultValueRefCountBase class 322
  defining in IDL 319
  eventtype keyword 318–319, 325, 330
  example 320
  factory 319–321, 323–324, 327
  implementing 321
  private keyword 320, 322
  public keyword 320, 322
  regular value types 328, 330
  state members 317
  truncatable keyword 330
  uses 318
  value box 330
  value type class 322
  value type factory 323
ValueFactoryBase class 323
valuetype keyword 317, 319, 325

value_project option 39, 41
value_template option 39, 41
ValueFactoryBase class 323
valuetype keyword
value types 317, 319, 325

Vanderbilt University xix, xxxix–xl, 3, 1449
See also DOC Group

vc6 build type 29, 36, 41, 44, 51, 62, 69
vc7 build type 36, 44, 51, 69
vc71 build type 29, 36, 51
vc8 build type 36
version option 39
versioned_so build flag 1379, 1387

Vinoski, Steve xxvii, xxxii, 215, 282, 1465, 1467

Visual C++ xxxi, 19, 27, 29–30, 33, 36, 1401
build libraries 1350, 1353, 1405
configure source code 1403
setting up environment 1403
verify build 1405

Vlissides, John 16, 404, 1465

VxWorks iv, 630, 776, 811, 1390, 1415–1417, 1429–1430, 1432
environment 1417
kernel and system configuration 1415
multicast routing 1416
using ACE and TAO 1415

W

wait set
data-centric publish-subscribe (DCPS) 1180

wait strategies 576–586, 766
wait-on-leader-follower 577, 581–583
wait-on-leader-follower-no-upcall 577, 583–586
wait-on-reactor 577, 581
wait-on-read 573–581

wait_for_completion parameter 534, 541–545
WaitForMultipleObjects() operation 403, 737
wait-on-leader-follower concurrency model 774
wait-on-leader-follower wait strategy 577, 581–583
wait-on-leader-follower-no-upcall wait strategy 583
wait-on-leader-follower-no-upcall wait strategy 577, 583–585
wait-on-reactor wait strategy 577, 581
wait-on-read wait strategy 573–580
Wang, Nanbor 1467
Washington University, St. Louis, Missouri xxvi, xxxix–xl, 3

See also DOC Group

wchar type codes 501, 703–704, 729
wfmo build flag 1379, 1387
wfmo reactor 540, 712, 737
Windows 21, 29, 611
Borland C++ 1409–1413
winregistry build flag 1379, 1388
work_pending() operation 154, 534, 536–538, 540–541, 544
WrongPolicy exception 272
wxWindows 815

X

X windows toolkit 707, 709
Xerces C++ 1345–1347, 1351
building 1346, 1350
XERCESCROOT environment variable 1331, 1346–1347, 1349, 1351–1352
XML service configurator 592, 603, 605
syntax 605
xt_reactor build flag 1378, 1383
XtAppContext 709