CORBA-III

Harald Gjermundrod

Original slides by:

Prof. David Bakken

Cpt. S 464/564 Lecture

Oct 5, 2004

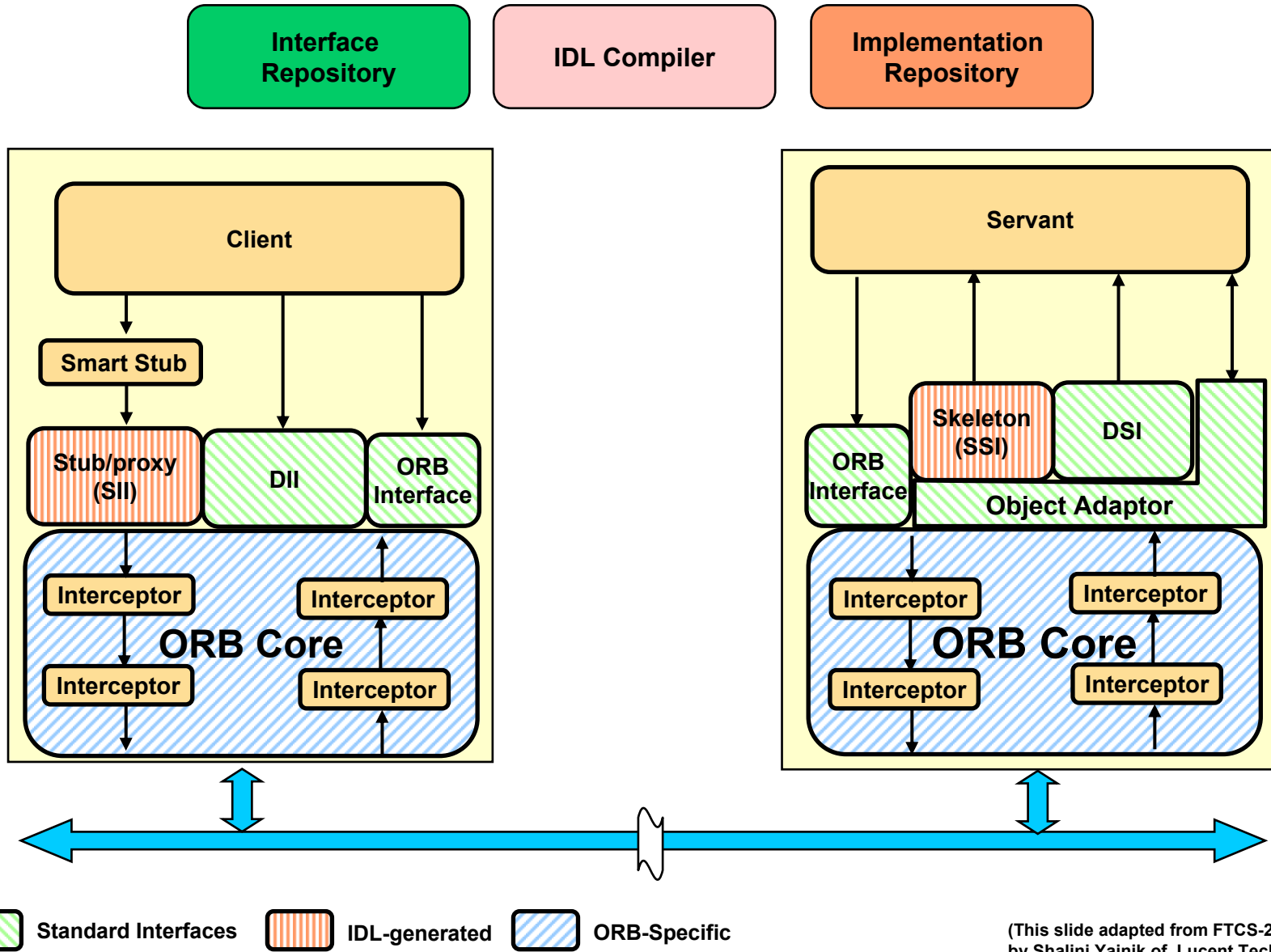
Administrative Items

- Today's topic: Overview of CORBA hooks and architectural components
  - CORBA Design Principles
  - CORBA Components
  - CORBA:Object
  - Static Invocation Interface (SII) Dynamic Invocation Interface (DII)
  - ORB core
  - GIOP and IIOP
  - Object References
  - User-defined exceptions
  - ORBACUS naming service
- Project #2 discussion

Major CORBA Design Principles

- Separation of interface and implementation
  - Clients depend on interfaces, not implementations
- Location transparency
  - Service use is orthogonal to service location
- Access transparency
  - Invoke CORBA objects just like local ones
- Typed interfaces
  - Object references are typed by interfaces
- Support of multiple inheritance of interfaces
  - Inheritance extends, evolves, and specialized behavior
  - Note: not inheritance of multiple implementations!
- Support of multiple interaction styles
  - Client/server
    - Some support for mobile code, too, with Objects by Value (OBV)
  - Peer processes
  - Publish/Subscribe (aka “push”)

CORBA Components and System Hooks



(This slide adapted from FTCS-29 Tutorial by Shalini Yajnik of Lucent Technologies)

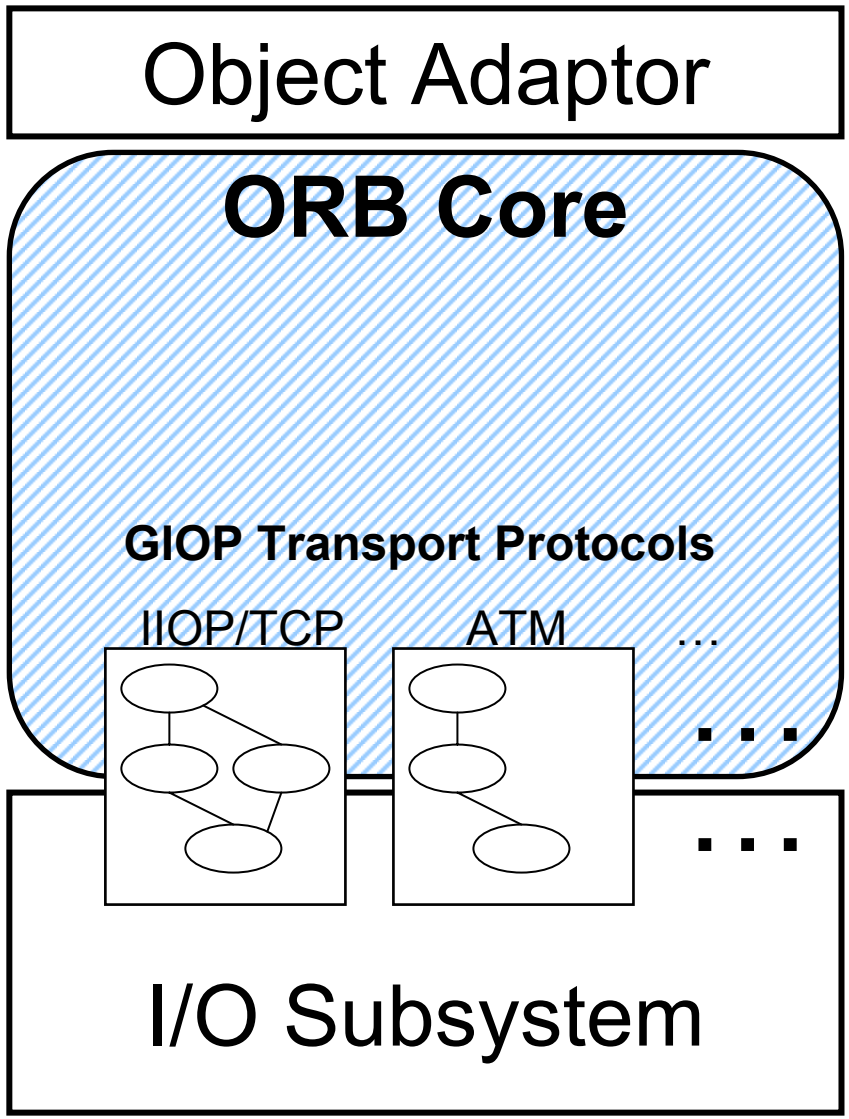
CORBA:Object class

- Base class for all proxies
- Useful utility methods:
  - `_is_a()`
  - `_is_equivalent()`
  - `_duplicate()// Not needed in Java`
  - `_release() // Not needed in Java`
  - `_is_local()`
  - `_is_remote()`
- Every call to `_duplicate` should be matched with a call to `_release`

SII and DII

- Static Invocation Interface (SII)
  - Most common way to use IDL
  - All operations specified in advance and known to client (by proxies/stubs) and server (by skeletons)
  - Simple
  - Typesafe
  - Efficient
- Dynamic Invocation Interface (DII)
  - Introspection
  - Less common way to use IDL
  - Lets clients invoke operations on objects whose IDL is not known to them at compile time (main advantage of DII)
    - Browsers of all sorts (interface browser, etc)
    - Debuggers
  - Clients construct a CORBA::Request (local) object, “pushing” arguments and operation name etc. on it like a stack
    - Exactly what a proxy does: same API to ORB Core
  - Also can use `send_deferred()` and `poll_response()`

ORB Core Overview



Features (server-side)

- Connection management
- Memory management
- Request transfer
- Endpoint demuxing
- Concurrency control

Other Features

- object_to_string() and string_to_object()
- Etc.

General Interoperability Protocol (GIOP)

Abstract protocol to allow for interoperability between different vendors' ORBs. To do this, it defines:

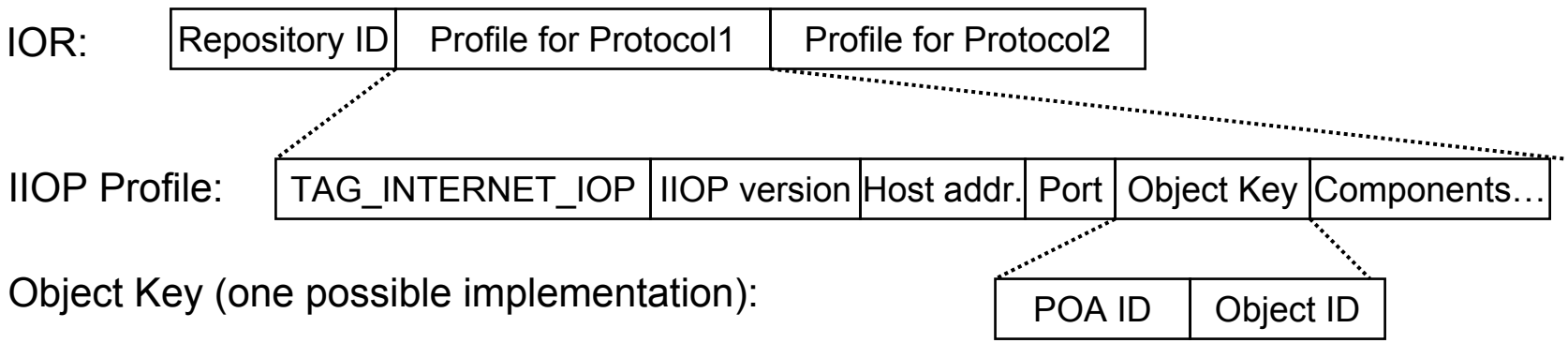
- Interoperable Object Reference (IOR) format
- Inter-ORB message for things like
  - Request
  - Reply
- Wire protocol (data transfer syntax): Common Data Representation (CDR). A coding for all IDL types, structured types, exceptions, object references. Covers coding into an octet stream, alignment boundaries, how to indicate byte ordering used.

IIOP

- IIOP is simply GIOP (an abstract protocol, remember) implemented over TCP/IP
- Must be implemented by every ORB
  - Gives a universal way for ORBs to communicate
  - A given ORB may implement different transports underneath GIOP, also
- CommunicationID = {IP address, port}

Object References

- Object reference
  - Opaque handle for client to use
  - Identifies exactly one CORBA object
  - IOR == “Interoperable Object Reference”
- References may be passed among processes on different hosts
  - As parameters or “stringified”
  - ORB will convert into form suitable for transmission over network
  - ORB on receiver side will create a proxy and return a pointer to it



- Object Key
  - Opaque to client
  - ORB-specific
- Object ID
  - Can be created by user or POA

User-Defined Exceptions

- CORBA has >20 pre-defined system exceptions that you should catch with every CORBA call of any kind
  - UNKNOWN, BAD_PARAM, NO_MEMORY, IMPL_LIMIT, COMM_FAILURE, INV_OBJREF, NO_PERMISSION, …
  - They all inherit from CORBA::Exception
  - Thrown by ORB implementation
- Programmers can also declare exceptions in IDL
  - Thrown by server side

Messenger IDL with Exceptions

- From the MessengerEx example
  - /net/niflab/orbacus/cs564/MessengerEx

```
// messenger.idl
interface Messenger  {
        exception ImNotHere {
            string reason;
        };
        boolean send_message (
                    in    string user_name,
                    in    string subject,
                    inout string message ) raises (ImNotHere);
};
```

Messenger Catching User-Level Exceptions

```
// MessengerClient.cpp
try {
    // …. Init ORB, get IOR from file or name service, make invocations …
}
catch( const Messenger::ImNotHere &inhEx ) {
    cerr << "Caught an ImNotHere exception: " << inhEx << endl;
    cerr << "reason: " << inhEx.reason << endl;
    return 1;
}
catch( const CORBA::COMM_FAILURE &commEx ) {
    cerr << "Caught a COMM_FAILURE: " << commEx << endl;
    return 1;
}
catch( const CORBA::Exception &ex) {…..}
```

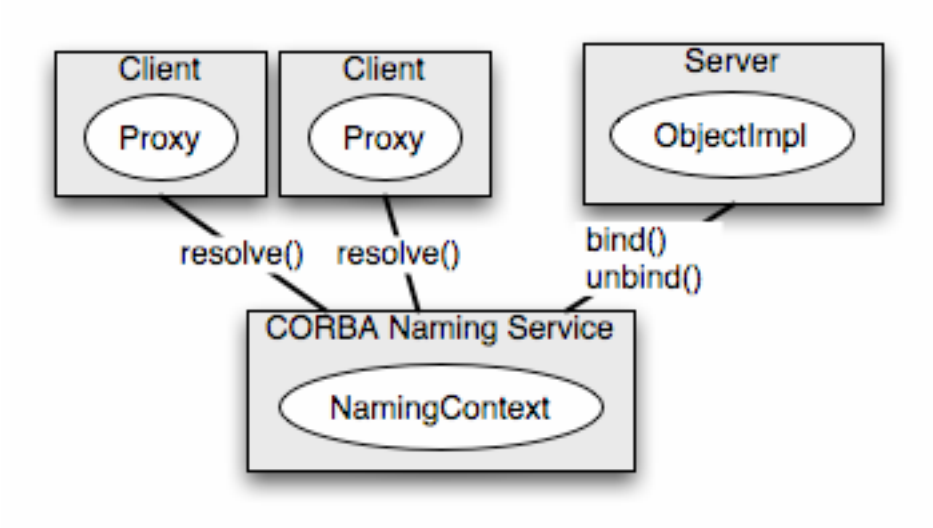
Messenger Throwing User-Level Exceptions

```
CORBA::Boolean
Messenger_impl::send_message(const char* user_name,
                             const char* subject, char*& message)
    throw(Messenger::ImNotHere,
          CORBA::SystemException)
{
 if (strcmp(user_name, "The_Professor") == 0) {
      // ...
} else if (strcmp(user_name, "Joe_Freshman") == 0) {
      // don't want to talk to a freshman--send him the ImNotHere exception
      // and tell him to talk to someone else
      cerr << "Throw ImNotHere" << endl;

      CORBA::String_var reason =
                              CORBA::string_dup("Go talk to someone else");
      throw Messenger::ImNotHere(reason);
} else {
            // …
```

CORBA Naming Service

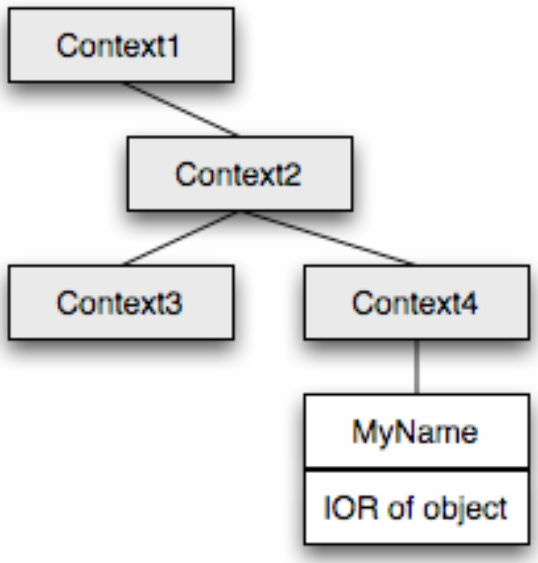
- Provides a mapping between a name and an object reference
- Names can be hierarchically structured by using contexts
- A Naming Server has operations to:
  - bind an object to a name, raises an exception if name is already bound
  - rebind an object to a name, overwrites the previous object reference
  - unbind the name from an object
- A client will
  - resolve an object given a name

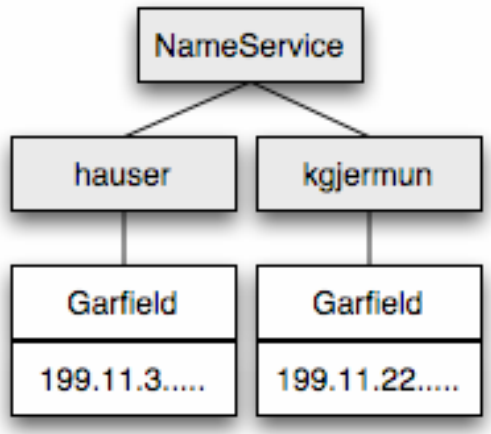


Naming Contexts

- To create a hierarchy in the Naming Service the server does:
  - bind_context() binds a new object of type NamingContext (a new directory)
  - rebind_context() again overwrites a NamingContext object
  - unbind will remove a name and its associated object reference

- In the figure a NamingContext object is bound to the name kgjermun
- In this context a object named Garfield is bound with its IOR
- When using the Naming Service make sure that you don't bind anything to another users NamingContext





Messenger Server Finding the Naming Service

- How do the different applications find the Naming Service
  - The call to CORBA::ORB_init takes arguments which can include the location of the naming service
  - The format of the argument is orb implementation dependent. :(
    - ORBacus, can given them as a switch to the program:
      - ./LibServer -ORBInitRef NameService=corbaloc::serverName:portNumber/NameService
      - ./LibServer -ORBInitRef NameService=corbaloc::nif-s1:26000/NameService
- Example of using namingservice from the Messenger example
  - /net/niflab/orbacus/cs564/MessengerName/

```
// 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() ) ) {
    cerr << "Nil Naming Context reference" << endl;
    throw 0;
} // … continued on next slide…
```

Messenger Server Binding a Context Obj

```
// Bind the example Naming Context, if necessary
 CosNaming::Name name;
 name.length( 1 );
 name[0].id = CORBA::string_dup( ”kgjermun" );
 try {
   CORBA::Object_var dummy = root->resolve( name );
 }
 catch ( const CosNaming::NamingContext::NotFound & ) {
       CosNaming::NamingContext_var dummy =
                                root->bind_new_context( name );
 }

 // … continued on next slide…
```

Messenger Server Binding a object

```
// 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() );
try {
  root->rebind( name, messenger_obj.in() );
}
catch ( const CosNaming::NamingContext::NotFound & ) {
  cerr << "Can't bind example/Messenger" << endl;
  throw 0;
}
```

Messenger Client Using the Naming Service

```
// … find naming service just like server did, same code
// Resolve the Messenger object
CosNaming::Name name;
name.length( 2 );
name[0].id = CORBA::string_dup( ”kgjermun" );
name[1].id = CORBA::string_dup( "Messenger" );

try { // NOTE THE TRY CATCH IS NOT IN THE EXAMPLE CODE
  CORBA::Object_var obj = root->resolve( name );
}
catch (const CosNaming::NamingContext::NotFound&)
{
  cerr << “Cannot find name” << endl;
  throw 0;
}
 // … continued on next slide…
```

Messenger Client Using the Naming Service

```
// Narrow
Messenger_var messenger = Messenger::_narrow( obj.in() );
if ( CORBA::is_nil( messenger.in() ) ) {
      cerr << "Not a Messenger reference" << endl;
      throw 0;
}
```

Tips when using the Naming Service

- If your server registers an object with the naming service, but doesn't unbind it, then the naming service still has the reference to the object
  - Example, stop the server with control-c
- If a client comes along and resolves this object, it will not notice that the server is not running until the client does a method call on this object
- Two ways to avoid the client from experiencing this problem:
  - Don't stop the server using control-c, but start a thread before calling orb->run() ; that will stop the servant and unbind the name before exiting
    - This is easy if you implement the project in Java, harder if you are using C++
  - When the client narrows the servant, call a dummy method. If the servant is not running, an exception will be thrown

- When you want the servant to be a client to another servant then don't create a new orb. You should pass something from the LibraryServer.cpp to your Library_impl.cpp so you don't need to create a new orb.

What to do when the unexpected happens

- What do you do when the following occur:

- Linking error
  - rm *.o
- Compile errors and you can not see what is wrong
  - rm *.o
- Your program compiles and runs, but something doesn't work that used to work. (99 out of 100 you did something wrong, but once in a while it is the compiler/make fault)
  - rm *.o

- Can add a clear or clean tag to your make file:

clean:

rm -f *.o

rm -f client

rm -f server

rm -f anything else that you would like to remove