Getting Started with MicroQoSCORBA

1. Introduction

MicroQoSCORBA, or MQC as it will be referred to in the rest of this document, is designed to provide embedded systems developers with a configurable middleware framework for developing distributed applications. This guide is intended to introduce distributed system developers to working with MQC. It provides installation instructions, a walkthrough of developing a simple MQC application, and an overview of the tools provided for configuring MQC.

2. Installation

System Requirements

The installation instructions in this document are designed for UNIX based operating systems (e.g. Linux, Mac OS X, etc.). However, the compiled jar files for MQC should run on any system with Java 2.

MQC has also been successfully run and tested on both SaJe, and TINI boards.

Installing MQC from a tar file

1. Unpack the MQC-RelXX.tgz into an appropriate directory (this creates a root directory called MQC).
2. The following environment variables need to be set for MQC to function properly:
   a) `JAVA_HOME=<path to java sdk>`
   b) `JAVA_BIN=<path to directory containing java, javac, javah, etc.>`
   c) `MQC_HOME=<path to the root directory of MQC>`
   d) `LD_LIBRARY_PATH=$MQC_HOME/jni` (this is not used on Mac OS X).

Installing MQC from CVS

1. The CVS repository for MQC resides on shakti.eecs.wsu.edu in /users/cvs.
2. Typically two environment variables will need to be set in order for cvs to be able to find MQC:
   a) `CVS_RSH="ssh"`
   b) `CVSROOT=":ext:username@shakti.eecs.wsu.edu:/users/cvs"`
3. Once the environment variables are set, change directory to the location that MQC is to be installed.
4. To check out the latest version of MQC into a directory named MQC enter:

   ```
   cvs checkout -P MQC
   ```

5. The following environment variables need to be set for MQC to function properly:
   a) `JAVA_HOME=<path to java sdk>`
   b) `JAVA_BIN=<path to directory containing java, javac, javah, etc.>`
   c) `MQC_HOME=<path to the root directory of MQC>`
   d) `LD_LIBRARY_PATH=$MQC_HOME/jni` (this is not used on Mac OS X).

6. Once the environment variables are set, enter:

   ```
   cd $MQC_HOME
   ```

7. To install MQC with the default Java millisecond timer enter:

   ```
   make all
   ```

   To install with system specific jni libraries that implement a microsecond timer enter:

   ```
   sudo make jni
   ```

   Note that sudo is only required on Mac OS X.

Installation Note

Both installation options require several environment variables to be set. It may be useful to consolidate these into a shell script that is called when the shell is loaded. An example csh script is shown in Figure 2.1 below.

```
#The following two lines are only required for the CVS install option
setenv CVS_RSH ssh
setenv CVSROOT :ext:username@shakti.eecs.wsu.edu:users/cvs

#Environment variables for MQC! - CSH
setenv JAVA_HOME   /usr/java/j2sdk1.4.1_03
setenv JAVA_BIN    /usr/java/j2sdk1.4.1_03/bin
setenv MQC_HOME    ~/MQC
setenv MQC_TOOLS   ~/MQCtools
if ("$?LD_LIBRARY_PATH") then
  setenv LD_LIBRARY_PATH $MQC_HOME/jni:$LD_LIBRARY_PATH
else
  setenv LD_LIBRARY_PATH $MQC_HOME/jni
endif
```

Figure 2.1: mqc.csh

3. Example Ping Application

The ping application distributed with MQC provides an example of a barebones MQC application. It is located in "$MQC_HOME/examples/ping", and the steps required to build this example are as follows:

Step 1: Creating an IDL file

An IDL file is a standardized way of defining the interfaces and objects that are utilized by CORBA. Included with the ping application is a file named "basic.idl", which can be seen in Figure 3.1.

```
module basic
{
   interface ping
   {
      boolean isAlive();
   };
};
```

Figure 3.1: basic.idl

The format of this file should be familiar to those who have worked with CORBA in the past.

Step 2: Running the MQC configuration tool

At the command prompt enter:

```
java -jar $MQC_HOME/CASE/MqcGui.jar
```

This will start the MQC configuration tool, which is designed to simplify the task of configuring MQC for a particular application. The tool begins on the "Data types" tab, which allows the user to specify the specific data types MQC will compile support into the ORB for. Make sure that only boolean is checked, as it is in Figure 3.2 below, because that is the only data type used by the ping application.

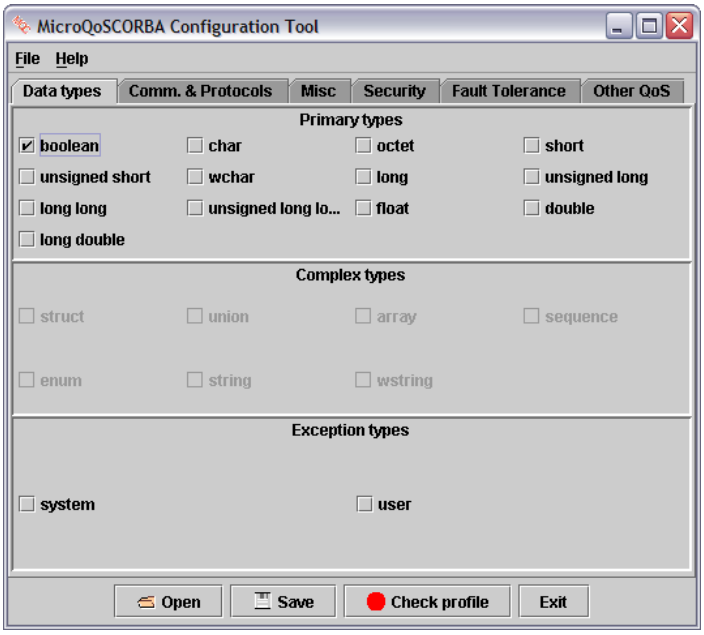

Figure 3.2: Configuration Tool

So, now click on the save button at the bottom of the panel. This will open a save file dialog. Save the file in the root directory of the ping application as a file named "config.mqcc", which is the default name for an MQC config file.

Note that the other tabs in this tool provide different quality of service (or QoS) options, but the defaults are fine for this application. A detailed description of this tool is beyond the scope of this document, but most of the options are fairly straightforward.

Step 3: Running the MQC IDL compiler

To simplify the various options for running the MQC IDL compiler a file named "Makefile.mqcc" is included with the ping application. Any future applications developed for MQC can use this file as a template for their own "Makefile.mqcc" file. The contents of this file are shown below in Figure 3.3.

```
JAVA=$(JAVA_BIN)/java
CASEDIR=$(MQC_HOME)/CASE

PACKAGE=basic
CONFIG_FILE=config.mqcc

all: idl
  $(MAKE) clean
  $(MAKE)

idl:
  $(JAVA) -jar $(CASEDIR)/MqcIdlCompiler.jar $(PACKAGE).idl $(CONFIG_FILE)

gui:
  $(JAVA) -jar $(MQC_HOME)/CASE/MqcGui.jar
```

Figure 3.3: Makefile.mqcc

To invoke the IDL compiler enter:

```
make -f Makefile.mqcc idl
```

Alternatively the IDL compiler can be invoked directly by entering:

```
java -jar $MQC_HOME/CASE/MqcIdlCompiler.jar basic.idl config.mqcc
```

After the IDL compiler runs, the stubs and skeletons are created and placed in a directory named "basic". As will be explained in the next section, an implementation specific file needs to be created.

Step 4: Implementing the interface

Some IDL compilers create a stubbed out implementation file, but MQC does not currently do this. Therefore it is necessary to implement the functionality provided by the `basic.ping` interface. As a convenience the root directory of the ping application contains a file named "pingImpl.java.example" which implements the functionality provided by the `basic.ping` interface. Its contents are shown in Figure 3.4.

```
public class pingImpl extends basic.pingPOA
{
   public boolean isAlive()
   {
      System.out.println("Pinged");
      return true;
   }
}
```

Figure 3.4: pingImpl.java.example

When implementing an IDL defined interface in Java, the modules in the IDL file map to packages in Java. Therefore, "basic.idl" (Figure 3.1) defines a Java package named `basic`, which contains an interface named `ping`. The `ping` interface contains a single method named `isAlive()`, which returns a boolean value.

Notice that the class `pingImpl` extends the class `basic.pingPOA`. The class `basic.pingPOA` is generated automatically by the MQC IDL compiler, and contains the code required for a portable object adapter (or POA).

Step 5: Writing the server

MQC's fine grained composibility is met in part by the use of macros. The macro preprocessor "m4" is used to include/exclude configuration specific code (e.g. OS type, Security QoS settings, etc.). Therefore both the client and server java code must be implemented in a .java.m4 file. See the advanced section for more details.

The file "Server.java.m4" in the root directory of the ping application implements a simple server. Its contents are shown in Figure 3.5 below.

```
import mqc.*;
import mqc.holders.*;
import java.io.*;

public class Server
{
   public static void main(String[] args)
   {
      S_ORB orb = new S_ORB();
      POA rootPOA = new POA(orb, "RootPOA");
      pingImpl pingObj = new pingImpl();

      Object object = rootPOA.servant_to_reference(pingObj);
      String corbaloc = orb.object_to_corbaloc(object);
      System.out.println("object to corbaloc!");
      System.out.println(corbaloc);
      try
      {
         FileWriter out = new FileWriter(new File("basic.corbaloc"));
         out.write(corbaloc);
         out.close();
      }
      catch(IOException e)
      {
         System.out.println("Error writing corbloc");
         return;
      }

      System.out.println("Waiting for requests...");
      orb.run();
   }
}
```

Figure 3.3: Server.java.m4

When run, the server creates an ORB, and a POA. An implementation of the ping interface is then associated with the "RootPOA". It then extracts a corbaloc identifier into a string. The corbaloc string is printed to the screen as well as written to a file named "basic.corbaloc". Finally, the server starts the orb.

Note that this example doesn't use any of the macros, but the auto generated Makefile requires a .java.m4 file.

Another thing to note is the use of a corbaloc string written to the screen and stored in a file to identify the ping object.

Step 6: Writing the client

Like the server, the file "Client.java.m4" contains a very simple client application. The source code for this application is shown in Figure 3.4.

```
import mqc.*;
import mqc.holders.*;

public class Client
{
   public static void main(String[] args)
   {
      Object object;
      C_ORB orb = new C_ORB();

      object = orb.corbaloc_to_object(args[0]);

      basic.ping pinger = basic.pingHelper.narrow(object);

      if(pinger.isAlive())
      {
         System.out.println("Ping successful");
      }
   }
}
```

Figure 3.4: Client.java.m4

This example creates an orb, and uses it to obtain the object specified by a corbaloc string that is passed as a command line argument to the program. It then invokes the isAlive() method, which prints "Ping successful" if true.

Step 7: Compilation

To complete the final step enter:

```
make jar
```

Running the Ping Example

A corbaloc string is a standardized way of encoding the information to locate a distributed CORBA object in a string. If both the client and server are run with a shared file system the client can find the server by way of the file "basic.corbaloc". If there is no shared file system, the corbaloc string can be copied manually when the server prints it to the screen.

To run the example on the same machine, simply open up two shells. Choose one to be the server, and one to be the client. On the server, enter:

```
java –jar Server.jar
```

When it is successfully running, a file called "basic.corbaloc" will appear in the ping directory containing the information the client will need to find the remote object on the server. On the client, enter:

```
java –jar Client.jar `cat basic.corbaloc`
```

The server should output “Pinged”, and the client should output “Ping successful” if everything was successful.

To run the example on different machines that do not share a file system, the only difference is in running the client. Instead of using `` `cat basic.corbaloc` `` to provide the client with the corbaloc string it must be entered directly like this:

```
java –jar Client.jar <corbaloc string copied from server's output>
```

4. Advanced Example Timing Application

There is an example application included with the MQC distribution located in a directory named “$MQC_HOME/examples/timing”. The timing application utilizes some of the more advanced features of MQC. For instance in Figure 4.1, a snippet of code from the timing example’s “Server.java.m4” shows that it has been written to utilize security macros in MQC.

```
S_ORB orb = new S_ORB();
__DEBUG__( System.out.println("ORB created!");)

__SECURITY__(
   System.out.println( "\n*** " + orb.cipherEncrypt.toString() + "  keyLen:
       __SEC_CIPHER_KEY_LEN__");
   // load the 'user' supplied key into the ORB
   byte[] orbKey = {10,11,12,13,14,15,16,17};
   try {
      orb.initCipherKey(orbKey);
   }
   catch (InvalidKeyException e) {
      System.out.println(e);
   }
)
__SECURITY_MD__(
   System.out.println("*** Using Message Digest: __SEC_MD_ALG__");
)
__SECURITY_MAC__(
   System.out.println("*** Using MAC: __SEC_MAC_ALG__
       (__SEC_MAC_ALG_PARAMS__)");
   // load the 'user' supplied MAC key into the ORB
   byte[] macKey = {20,21,22,23,24,25,26,27};
   try {
      orb.initMacKey(macKey);
   }
   catch (InvalidKeyException e) {
      System.out.println(e);
   }
)
```

Figure 4.1: Excerpt from Server.java.m4

The m4 macros used in this example can be identified by the double underscores that precede and follow each macro name (e.g. “__SECURITY__”). This example shows that it is possible to use the same server code with different QoS property configurations. The various security options are only built into the server if MQC is configured to use those options.

5. MQC Directories of Note

$MQC_HOME/CASE

This directory contains various tools for MQC. It contains the MQC IDL compiler, as well as the MQC configuration GUI.

$MQC_HOME/examples

This directory contains different MQC examples. Of particular note are the "ping", and "timing" examples, which were referred to in this document.

$MQC_HOME/examples/config

This particular directory contains several makefiles, which generate some default MQC "config.mqcc" files. This provides a way to obtain a "config.mqcc" file with out running the MQC configuration GUI.

$MQC_HOME/mqc

This directory contains the source files for the MQC ORB.

$MQC_HOME/mqc/protocols

This directory contains the source files for the protocols that MQC implements.

$MQC_HOME/mqc/security

All of the MQC security source files are found under this directory.

$MQC_HOME/mqc/transports

The source files in this directory implement the various transport protocols that MQC supports.