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 $TAO_ROOT/DevGuideExamples/GettingStarted.

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.

    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.

    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.
```
Getting Started

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;
}

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 send_message() operation.

Create MessengerClient.cpp with the following contents:

#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:

```plaintext
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 *portablesserver* 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:

  ```bash
  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:
  
  ```
  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:
  
  ```
  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:
  
  ```
  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!