Artix™

Artix Demo Guide

Version 1.3, December 2003

Making Software Work Together™

Updated: 01-Mar-2004

M 3 1 1 5

Contents

Preface

About this Guide

Artix can be used in many different ways. It can be used to:

- Code C++ Web service client applications that run against distributed services.
- Develop and deploy a C++ Web service.
- Create a switch or bridge between two applications based on different middleware products, e.g., TIBCO Rendezvous and WebSphere MQ.

This document, and the accompanying coding examples, will teach you about each of these types of applications and demonstrate the functionality included in the Artix™ product.

Audience

This guide is aimed at new users of Artix who wish to see examples of Artix in action, and gain an idea of the various capabilities of the product.

Related documentation

This guide assume you have read the following document:

Artix Getting Started

This guide also refers you to the following documents for more detail on the GUI and programming issues respectively:

Artix User's Guide

Artix Programmer's Guide

Organization of this guide

This guide is divided as follows:

Chapter 1 gives a brief overview of the functionality of Artix using the spellcheck service deployed by Google™.

Chapter 2, Chapter 3, and Chapter 5 through Chapter 9 illustrate how to use different transports for the same client and server code:

- In Chapter 2 and Chapter 3, you will first develop a simple C++ client, which uses SOAP over HTTP, to invoke on a Web service.
- In Chapter 5 you will change the transport from HTTP to IIOP.
- In Chapter 6 you will change the transport to Tuxedo.
- In Chapter 7 you will change the transport to WebSphere™ MQ.
- In Chapter 8 you will use the TIBCO encoding and the TIBCO Rendezvous transport.

- In Chapter 9 you will use CORBA encoding and IIOP transport.

Chapter 4 introduces Artix configuration principles and includes an elementary discussion of runtime configuration settings.

In Chapter 10 you will be introduced to routing, the ability to propagate a request using multiple transport protocols.

Chapter 11 describes how you can use an Artix server can receive requests over multiple protocols and pass the invocation to a common implementation object.

Chapter 12 describes how you can specify that operations use oneway semantics.

Finally, Chapter 13 delves into code generation, using complex data types in your application code, and presents a comparison between the code generated from SOAP/RPC and document/literal encoded WSDL files.

Note: The step-by-step instructions and PATHs to specific directories and files are presented in Windows format. The demo code will run on UNIX systems, and `makefile`s and scripts to set required environment variables are provided for both Windows and UNIX operating systems. If you want to run these demos on a UNIX system, you are responsible for transposing Windows syntax into UNIX syntax.

Additional related resources

The IONA knowledge base contains helpful articles, written by IONA experts, about Orbix and other products. You can access the knowledge base at the following location:

http://www.iona.com/support/kb/

The IONA update center contains the latest releases and patches for IONA products:

http://www.iona.com/support/update/

If you need help with this or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `doc-feedback@iona.com`.

Typographical conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, Windows 95, or Windows 98 command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Introduction

This chapter uses the coding example in one of the product demos to introduce you to the mechanics of working with Artix.

In this chapter

This chapter discusses the following topics:

A Sample Artix Application

The SpellCheck Demo

In this example, Artix is used to code a C++ Web service client that runs against the spellcheck service deployed by Google™.

The WSDL file

One of the interesting aspects about this example is that it does not require a local copy of the WSDL file describing the service. This file is accessed remotely and used by the Artix code generation utility to create the necessary stub and helper classes. Consequently, this example represents the simplest approach to using Artix.

When you run the Artix code generation utility – `wsdltocpp` – the WSDL file is accessed remotely from the site `api.google.com/GoogleSearch.wsdl`. The code generation utility uses information from the WSDL file to create the classes your client application needs to contact the Web service. You then write a client application that uses these classes.

Point your Web browser to `http://api.google.com/GoogleSearch.wsdl` to display the WSDL file that describes the Web service. This service offers three operations, which are described within the `<portType>...</portType>` tags. In this example, the client application is completely coded and uses the `doSpellingSuggestion` operation. However, you can extend this application to use other operations.

Notice that the `url` for the actual Web service, provided within the `<service>...</service>` tags, is different from the `url` through which you view the WSDL file. The Google Web service is available at `http://api.google.com/search/beta2`. When the client application runs, the stub/proxy code accesses the WSDL file again and extracts the service's actual `url`.

Within the WSDL file, the `portType` section is particularly significant. The `portType` is conceptually identical to a J2EE or CORBA interface definition; it describes the operations available on an "object." When the `wsdltocpp` code generation utility creates application classes, it includes the value of

the portType name attribute within the names of the classes. In the Google WSDL, the portType name is GoogleSearchPort, and the names of the generated classes all begin with GoogleSearchPort.

Note: You may give your portType any name. The wsdltocpp code generation utility does, however, modify some names. If the name you select ends with PortType, such as HelloWorldPortType, the code generation utility strips PortType from the names of the generated classes. The generated class names will start with HelloWorld rather than HelloWorldPortType.

In more involved WSDL files, there may be multiple portType sections. When running the wsdltocpp code generation utility you will use command line parameters to specify which portType (and service) should be represented by the generated code. This concept is developed more completely in later chapters.

The makefile

Since there is a straight-forward logic to the naming convention for the generated code, you will be able to write or edit your makefiles as required. The makefile for this sample is complete. You can review its contents to understand more completely the compilation and linking processes.

Basically building the application is a three-step process:

1. Use the WSDL file to generate the required stubs and helper classes.
2. Compile the application source code and the generated classes.
3. Link the compiled code to the required Artix libraries.

The GoogleSearchPortClient class

This generated class represents the stub, or proxy, to the distributed Web service; its declaration and implementation are described in the files GoogleSearchPortClient.h and GoogleSearchPortClient.cxx.

Your client application creates an instance of this class and then uses it to invoke the desired Web service operation. Later chapters of this guide will discuss this class in greater detail. To understand this example, you need only appreciate the fact that this class includes the method doSpellingSuggestion, which corresponds to the Web service operation used by the client application.

The client application

This application is fully coded and included in the file `spellcheck.cxx`. The application is quite simple: it obtains input from the command line, submits a word for spell checking to the Web service, and displays the corrected spelling. The significant code fragment is:

```
GoogleSearchPortClient the_client; // create stub/proxy instance
    ...
    // invoke the method
the_client.doSpellingSuggestion(user_key, phrase, result);
    ...
```

The parameters `user_key`, `phrase`, and `result` are string variables representing the Google license key, the input word, and the corrected spelling returned from the Web service. Note that Artix method invocations use out parameters rather than return values to return data from the service to the client application.

Compiling and Running the Sample Application

Firewall restrictions

To successfully build and run this application, you must have access to the public Internet. If your computer is behind a firewall that prevents Internet access, you will not be able to build and run this example. The other examples described in this guide are completely self-contained and will not be affected by firewall restrictions.

Note: Throughout this guide, compilation instructions are presented that assume you have not already built the sample applications. You can build the entire suite of sample applications from the `<installationDirectory>\artix\1.3\demos` directory. Alternatively, you can build each sample application from its own directory, as described in this guide.

Compiling the Application Source Code

You must first set your environment to insure that the Artix libraries and executables are accessible.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\spellcheck` directory.
3. In the command window, issue the command

   ```
   nmake all
   ```

Makefile processing

The `makefile` runs the batch file `wsdltocpp.bat` with the command,

```
wsdltocpp.bat
    -w "http://api.google.com/GoogleSearch.wsdl"
    -n GOOGLE
```

which runs the code generation utility; then the source code files are compiled and linked into the executable `client.exe`.

Running the Application

From the command window used in the previous section, issue the command

```
client[.exe] <someMisspelledWord>
```

The correct spelling is displayed and the process ends.

Alternatively, issue the command

```
client[.exe]
```

followed by the enter key. Then type a misspelled word and press the enter key again.

```
<someMisspelledWord>
```

The correct spelling is displayed and the process ends.

CHAPTER 2

Developing a Web Service Client

In this chapter, you will develop an Artix™ C++ client application. In the following chapter, you will code an Artix C++ server application.

In this chapter

This chapter discusses the following topics:

Coding the Artix C++ Web Service Client

As a client application developer, your only information about the target Web service comes from the WSDL file. Artix includes a utility that reads the WSDL file and generates the client-side stubs (and server-side skeletons) that you will use in coding your client application.

This chapter discusses the support Artix provides to the client application developer. The server application will be compiled and run as part of the following chapter.

Generating the Stub Code

The `makefile` in the `<installationDirectory>\artix\1.3\demos\hello_world\http_soap\client` directory specifies the WSDL file used to generate the stub and skeleton code. Although the skeleton code is not needed to code a client application, there is no harm in generating this code. For this discussion, you do not need to be concerned with the skeleton code.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\http_soap\client` directory.
3. Use a text editor to open the `makefile`. Observe that the `$(WSDL)` variable now includes the relative path to the local WSDL file.
4. Close the file.
5. In the command window, issue the command

   ```
   nmake all
   ```

Makefile processing

The `makefile` runs the batch file `wsdltocpp.bat` with the command,

```
wsdltocpp.bat -n HW $(WSDL)
```

The class `xmlbus.WSDLToCPPClient` generates the stub and skeleton classes. The `$(WSDL)` tag resolves to the file location for the WSDL file describing the Web service you create and deploy in the following chapter.

```
WSDL=HelloWorld.wsdl
```

Command-line arguments

Although the `makefile` includes the proper commands for this application, the `wsdltocpp.bat` file may use any, or all, of the following command line arguments; only the WSDL-URL argument is required.

```
[-e Web-service-name] [-t port] [-b binding-name]
[-i portTypeName] [-d output-directory] [-n namespace]
[-impl] [-m {NMAKE | UNIX}] [-f] [-server] [-client] [-sample]
[-v] [-license] [-all] [-flags]
WSDL-URL
```

This example uses one command line argument:

- `-n` specifies a C++ namespace for the generated source code.

Generated files

The following files are created from the WSDL file:

Table 1: *Files generated by wsdltocpp utility*

Generated files	Description
`HelloWorld.h`	Describes the class that represents the Web service API. This class is the superclass for both client stubs and server skeleton classes.
`HelloWorldClient.h`, `HelloWorldClient.cxx`	Client-side stub code
`HelloWorldServer.h`, `HelloWorldServer.cxx`	Server-side skeleton code
`HelloWorldTypes.h`, `HelloWorldTypes.cxx`	C++ class descriptions of complex data types defined in the WSDL file. In this example no complex data types are defined, so these files are not generated for this demo.

Using the `-client` command line parameter would suppress generation of the `HelloWorldServer.h` and `HelloWorldServer.cxx` files.

HelloWorld.h, HelloWorldTypes.h

The `HelloWorld.h` and `HelloWorldTypes.h` contain all the information you need to write client code.

For this simple demo, the `HelloWorldTypes.h` file is not generated.

HelloWorld.h file

The generated content of the `HelloWorld.h` file is:

```
#include <it_bus/bus.h>
#include <it_bus/types.h>
#include "HelloWorldTypes.h"

namespace HW
{
    class HelloWorld
    {
        public:
        HelloWorld() {}
       ~HelloWorld() {}

        virtual void greetMe
          (const IT_Bus::String& stringParam0,
           IT_Bus::String& var_return)
        IT_THROW_DECL((IT_Bus::Exception)) = 0;

        virtual void sayHi(IT_Bus::String& var_return)
        IT_THROW_DECL((IT_Bus::Exception)) = 0;
    };
};
```

The `HelloWorld` class is the superclass for classes `HelloWorldClient` and `HelloWorldServer`, and so provides a single service-oriented API for both client and server processes.

The last parameter in each method declaration represents the method's return value.

HelloWorldClient.h, HelloWorldClient.cxx

These files contain client stub code. You do not work directly with the code in these files. However, the client application instantiates instances of the `HelloWorldClient` class, so you should be familiar with its constructor methods.

HelloWorldClient.h file

- For all of the constructors, the `IT_Bus::Bus_ptr` parameter has an assigned default value. You do not need to provide this value.
- The overloaded constructors take parameters let you specify a different location for the WSDL file, service name and port name.

```
#include "HelloWorld.h"
#include <it_bus/service.h>
#include <it_bus/bus.h>
#include <it_bus/types.h>

namespace HW
{
    class HelloWorldClient :
      public HelloWorld, public IT_Bus::ClientProxyBase
    {

      private:
        IT_Bus::Bus_var    m_bus;
        IT_Bus::Service *  m_service;
        IT_Bus::String     m_port_name;
        IT_Bus::Port *     m_port;

      public:
        HelloWorldClient(
            IT_Bus::Bus_ptr bus = 0
        );

        HelloWorldClient(
            const IT_Bus::String & wsdl,
            IT_Bus::Bus_ptr bus = 0
        );

        HelloWorldClient(
            const IT_Bus::String & wsdl,
            const IT_Bus::QName & service_name,
            const IT_Bus::String & port_name,
            IT_Bus::Bus_ptr bus = 0
        );
```

```
        HelloWorldClient(
            const IT_Bus::Reference & reference,
            IT_Bus::Bus_ptr bus = 0
        );

        ~HelloWorldClient();

        virtual void
        greetMe(
            const IT_Bus::String & stringParam0,
            IT_Bus::String & var_return
        ) IT_THROW_DECL((IT_Bus::Exception));

        virtual void
        sayHi(
            IT_Bus::String & var_return
        ) IT_THROW_DECL((IT_Bus::Exception));

    };
};
```

HelloWorldClient.cxx file

In this example, the client application uses the single argument constructor without providing a value for the `IT_Bus::Bus_ptr` parameter. This constructor calls service factory method `create_service()`, whose first argument specifies the WSDL file's URL.

```
HelloWorldClient::HelloWorldClient(
    IT_Bus::Bus_ptr bus
)
{
    if (bus == 0)
    {
        m_bus = IT_Bus::Bus::create_reference();
    }
    else
    {
        m_bus = IT_Bus::Bus::_duplicate(bus);
    }
```

```
    m_service =
   ServiceFactory::get_instance(m_bus.get()).create_service(
       "../common/HelloWorld.wsdl",
       QName("", "HelloWorldService",
                 "http://xmlbus.com/HelloWorld")
    );
    m_port_name = "HelloWorldPort";
    m_port = m_service->create_port(m_port_name);
}
```

Writing the Client Application Code

The following code shows a client application that invokes Web service methods `sayHi` and `greetMe` on an instance of the `HelloWorldClient` class. Artix does not create starting point code for the client application.

Client.cxx file

The client application includes the following code:

```
#include <it_bus/bus.h>
#include <it_afc/Exception.h>
#include <it_cal/iostream.h>

#include "../common/HelloWorldClient.h"

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

using namespace HW;

int
main(
    int argc,
    char* argv[]
)
{
    cout << "HelloWorld Client" << endl;

    try
    {
        IT_Bus::init(argc, argv);
        HelloWorldClient hw;

        String string_in;
        String string_out;

        hw.sayHi(string_out);
        cout << "sayHi method returned: "
             << string_out << endl;
```

```
   if (argc > 1) {
          string_in = argv[1];
        } else {
          string_in = "Early Adopter";
        }

        hw.greetMe(string_in, string_out);
        cout << "greetMe method returned: "
             << string_out << endl;
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Caught Unexpected Exception: "
             << endl << e.Message()
             << endl;
        return -1;
    }
    return 0;
}
```

Compiling the Client Application

Running the `makefile` as described earlier generates all of the helper files and compiles and links the client application.

Running the client application

You will not be able to run this application until you complete coding the server process as described in the next chapter.

CHAPTER 3

Developing a Web Service Server

In this chapter, you will develop an Artix™ C++ server application.

In this chapter

This chapter discusses the following topic:

Coding the Artix C++ Web Service Server

As a server application developer, your only information about the Web service comes from the WSDL file. Artix includes a utility that reads the WSDL file and generates the server-side skeletons and starting point code for your implementation object.

Generating the Skeleton and Starting Point Implementation Code

The `makefile` in the `<installationDirectory>\artix\1.3\demos\hello_world\http_soap\server` directory creates the skeleton class code. For this first example, the starting point code for the implementation object has been provided. However, you could use the `-impl` command line argument when running the `wsdltocpp` utility, which would generate starting point code.

The following additional files are created in response to the `-impl` command line argument.

Table 2: *Generated Files*

Generated files	Description
`HelloWorldImpl.h`, `HelloWorldImpl.cxx`	Generated by the `-impl` flag, these files contain starting point code for the target object that provides the Web service functionality

HelloWorldImpl.h and HelloWorldImpl.cxx

The header file includes definitions for two classes:

- `HelloWorldImpl` class, which is your Web service implementation
- `HelloWorldImplFactory` class

HelloWorldImpl.h file

This file is complete and does not require any customization or extension.

```
#include "HelloWorldServer.h"

class HelloWorldImpl :
  public HW::HelloWorldServer
  {
    public:
      HelloWorldImpl(IT_Bus::Bus_ptr bus, IT_Bus::Port *port);
      ~HelloWorldImpl();
```

```
virtual void
      greetMe(
          const IT_Bus::String & stringParam0,
          IT_Bus::String & var_return
      ) IT_THROW_DECL((IT_Bus::Exception));

      virtual void
      sayHi(
          IT_Bus::String & var_return
      ) IT_THROW_DECL((IT_Bus::Exception));

  };

class HelloWorldImplFactory :
  public IT_Bus::ServerFactoryBase
  {
    public:
      HelloWorldImplFactory();
      virtual ~HelloWorldImplFactory();

      virtual IT_Bus::ServerStubBase*
      create_server(IT_Bus::Bus_ptr bus, IT_Bus::Port *port);

      virtual const IT_Bus::String &
      get_wsdl_location();

      virtual void
      destroy_server(IT_Bus::ServerStubBase* server);

    private:
      IT_Bus::String m_wsdl_location;
  };
```

HelloWorldImpl.cxx file

The generated code includes empty method bodies. You add your processing logic to the method bodies in this implementation file.

```
#include "HelloWorldImpl.h"
#include <it_cal/cal.h>

HelloWorldImpl::HelloWorldImpl()
{
}
```

```
HelloWorldImpl::~HelloWorldImpl()
{
}

void HelloWorldImpl::greetMe
 (const IT_Bus::String& stringParam0,
  IT_Bus::String& var_return)
IT_THROW_DECL((IT_Bus::Exception))
{
}

void HelloWorldImpl::sayHi(IT_Bus::String& var_return)
IT_THROW_DECL((IT_Bus::Exception))
{
}
```

Writing the Web Service Implementation Code

If desired, the `wsdltocpp` utility creates starting point source code files for your implementation class. These files are generated into the same directory as the stub, skeleton, and helper class files.

The `HelloWorldImpl.h` and `HelloWorldImpl.cxx` files contain properly formatted method declarations and fully functional factory class methods, but you must add the processing logic for each of your Web service's methods. For the HelloWorld Web service, you must complete the coding of the `greetMe` and `sayHi` methods. However, for this demo, the coding within the `HelloWorldImpl.cxx` file is already complete.

sayHi method

The `sayHi` method simply returns a message to the client application. Complete this method body by adding the following two lines of code:

```
cout << "HelloWorldImpl::sayHi called" << endl;
var_return = IT_Bus::String
     ("Greetings from the Artix HelloWorld Server");
```

greetMe method

The `greetMe` method returns a message that includes the input parameter. Complete this method by adding the following two lines of code:

```
cout << "HelloWorldImpl::greetMe called with message: "
     << stringParam0 << endl;
var_return = IT_Bus::String
     ("Hello Artix User: ") + stringParam0;
```

Required namespace declarations

To enable `std::cout`, and to simplify your coding, you must also add the following declarations to the `HelloWorldImpl.cxx` file:

```
IT_USING_NAMESPACE_STD
using namespace IT_Bus;
```

Writing the Server Mainline

The server mainline is quite simple and basically unchanged for all of the demos discussed in this document.

Server.cxx file

The server mainline process includes the following code:

```
#include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_bus/fault_exception.h>

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

int
main(
    int argc,
    char* argv[]
)
{
    cout << "HelloWorld Server" << endl;

    try
    {
        IT_Bus::init(argc, argv);
        IT_Bus::run();
    }
    catch (IT_Bus::Exception& e)
    {
        cout << "Error occurred: " << e.Error() << endl;
        return -1;
    }
    return 0;
}
```

This file is complete and does not require editing.

Compiling the Server Application Code

Running the `makefile` as described in Generating the Skeleton and Starting Point Implementation Code generates all of the helper files and compiles and links the server application. However, if you add or change the business logic in your implementation class, you will need to recompile and create a new server process.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\http_soap\server` directory.
3. Use a text editor to open the `makefile`. Observe that the `$(WSDL)` variable now includes the relative path to the local WSDL file.
4. Close the file.
5. In the command window, issue the command

   ```
   nmake all
   ```

Running the Artix C++ Web Service Application

You must first start the server process and then run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `artix_env[.bat]` file.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\http_soap\server` subdirectory and issue the command:

   ```
   start server
   ```

3. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\http_soap\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows.

Terminating the server process

Issue the `Ctrl-C` command in the corresponding command window.

CHAPTER 4

Configuring Artix™

This chapter briefly introduces the process of configuring an Artix installation. Configuration includes two aspects: setting up the host computer environment; and setting up the common and application specific runtime environment.

Also included in this chapter is a discussion of application logging, which, as you will learn, is set through configuration rather than through coding.

In this chapter

This chapter discusses the following topics:

Establishing the Host Computer Environment

The host computer's environment is configured during both the installation process and through running the script `artix_env.bat`, which is created during the installation process.

Configuration During Installation

Prior to running the Artix installation procedure, you specified the path to your JDK by setting the variable `JAVA_HOME`. Generally, this variable is set within the system environment, but you could set it only within the command window used to run the installer.

During the installation process, you accept, or specify, various environment settings, for example, the installation directory or whether to create/update existing system environment variables (`IT_PRODUCT_DIR` and `PATH`).

The Artix installer uses your input to write the contents of the script file `artix_env.bat`. The installer may also create/update the `IT_PRODUCT_DIR` and `PATH` system variables.

If you followed the standard product installation, the system environment variables will be set for all users of your computer and your `artix_env.bat` file will include the following commands.

The artix_env.bat file

Note that this file sets the `IT_PRODUCT_DIR` and `PATH` environment variables, if necessary. These are the system variables that could be set by the installer, so there is actually no requirement that you accept the installer's offer to set these variables.

By placing the Artix `bin` directories first on the `PATH`, this script insures that the proper libraries, configuration files, and utilities, e.g., IDL compiler, are used. Consequently, there should not be any problems if Orbix and/or Tuxedo (both of which include IDL compilers and CORBA class libraries) are installed on your host computer.

Note: The Orbix environment script, `<domainName>_env.bat`, also sets the `PATH`, `IT_PRODUCT_DIR` and other product specific environment variables to values appropriate to the installation. Consequently, there should not be any problems running this product on a computer that also hosts Artix and/or Tuxedo.

Running the artix_env.bat Script

You must set the Artix environment in each command window. All of the environment settings required by Artix are set by running the script `artix_env.bat`.

Depending on other environment settings, you may need to set environment variables for the Microsoft Visual C++ compiler. This is accomplished by running the script `vcvars32.bat`, which is located in the `...Microsoft Visual Studio\VC98\bin` directory.

Establishing the Runtime Environment

Artix is built upon IONA's Adaptive Runtime Architecture (ART). Runtime behaviors are established through common and application specific configuration settings that are applied during application startup. As a result, the same application code, without changes, may be run under varied configuration environments.

With Artix, runtime configuration values are maintained in a configuration file named `artix.cfg`, which is found in the directory `<installationDirectory>\artix\1.3\etc\domains`. For many of the demos you do not need to edit any of the entries in this configuration file. For some of the later demos, you will be editing/extending the contents of this file.

The orb_plugins Configuration Value

One of the configuration values that you will review most frequently is `orb_plugins`. This variable is a list of runtime plugins – code libraries – that should be loaded during application startup. It is through the `orb_plugins` entry that you specify what transports, logging paradigms, or high level middleware switching functionality will be available to an Artix process.

The default entry for the `orb_plugins` variable includes the commonly used logging and transport plugins.

Global orb_plugins value

The default value for the `orb_plugins` configuration entry is defined within the global scope of the `artix_env.bat` file.

```
orb_plugins = [
  “xmlfile_log_stream”,
  “iiop_profile”,
  “giop”,
  “iiop”,
  “soap”,
  “http”,
  “tunnel”,
  “ws_orb”
];
```

This listing is suitable for Artix applications that use SOAP/HTTP, SOAP/IIOP_Tunneling, and CORBA/IIOP transports. In later demos you will edit this listing so that the WebSphere MQ, Tuxedo or TIBCO Rendezvous™ transports can be used and so that message requests can be routed from one transport to another.

Configuration Scopes

Application specific configuration variables either override default values assigned to common configuration variables or establish new configuration variables. Configuration scopes are localized through a name tag and delimited by a set of curly braces terminated with a semicolon (`nameTag {…};`). Additionally, a configuration scope may include nested configuration scopes. Configuration variables set within nested configuration scopes take precedence over values set in enclosing configuration scopes.

In the `artix.cfg` file, there are several predefined configuration scopes. For example, the `demo` configuration scope includes nested configuration scopes for some of the examples included with the product.

Demo configuration scope

Within the `artix.cfg` file, configuration scopes are defined for many of the product demos.

```
demo
{

 fml_plugin
 {
     orb_plugins = ["local_log_stream",
            "iiop_profile", "giop", "iiop",
            "soap", "http", "G2", "tunnel",
            "mq", "ws_orb", "fml"];

 };

 telco
 {
     orb_plugins = ["local_log_stream",
            "iiop_profile", "giop", "iiop",
            "G2", "tunnel"];
     plugins:tunnel:iiop:port = "55002";
     poa:MyTunnel:direct_persistent = "true";
     poa:MyTunnel:well_known_address =
            "plugins:tunnel";
```

```
server
    {
        orb_plugins = ["iiop_profile", "giop",
           "iiop", "ots", "soap", "http",
           "G2", "tunnel"];
       plugins:tunnel:poa_name = "MyTunnel";
    };
};

switch
{
    orb_plugins = ["xmlfile_log_stream",
           "iiop_profile", "giop", "iiop",
           "soap", "http", "mq", "ws_orb",
           "interopbase_typefactory",
           "routing"];

    event_log:filters = ["*=FATAL+ERROR"];

    plugins:routing:wsdl_url=
           "InteropBase.wsdl";

    plugins:interopbase_typefactory:
     shlib_name =
           "it_demo_switch_type_factory";
};

no_switch
{
    orb_plugins = ["xmlfile_log_stream",
           "iiop_profile", "giop", "iiop",
           "soap", "mq", "ws_orb",
           "interopbase_typefactory"];

    event_log:filters = ["*=FATAL+ERROR"];

    plugins:interopbase_typefactory:
     shlib_name =
           "it_demo_switch_type_factory";
};
```

```
 tibrv
 {
     orb_plugins = ["local_log_stream",
            "iiop_profile", "giop", "iiop",
            "soap", "http", "tibrv"];

     event_log:filters = ["*=FATAL+ERROR"];
 };
};
```

Note how the `orb_plugins` list is redefined within each configuration scope. Demos within this document will refer to some of these configuration scopes and you will be directed to define additional configuration scopes.

Using Configuration Scopes

Generally when you create a new configuration scope you will force an Artix process to run under the configuration by supplying an `ORBname` parameter to the `IT_Bus::init` method invocation. During process initialization, Artix searches for a configuration scope with the same name as the `ORBname` parameter. For example, to start an Artix process under the configuration specified in the `demo.tibrv` configuration scope, your application would include code similar to the following fragment.

```
IT_Bus::init (argc, argv, “demo.tibrv”);
```

If a corresponding configuration scope is not located, the process starts under the higher level configuration scope. If there are no configuration scopes corresponding to the `ORBname` parameter, the Artix process runs under the default global scope. For example, if the nested configuration scope `tibrv` does not exist, the process would start using the configuration specified in the `demo` configuration scope; if the scope `demo` does not exist, the process runs under the default global scope.

Rather than supplying an `ORBname` parameter within your source code, you may include this information as a command line argument when starting an Artix process. Initialization values specified as command line arguments take precedence over corresponding entries in the source code or system environment.

For example, an application using the following initialization syntax

```
IT_Bus::init (argc, argv);
```

will run using the `ORBname` and configuration scope `demo.tibrv` when the following command is used to start the process.

```
<processName>.exe [application parameters] -ORBname demo.tibrv
```

In following demos, you will use both the coding and command line techniques to run your applications under modified configurations.

Note: The ordering of application parameters and initialization arguments does not matter provided you invoke `IT_Bus::init` before beginning your application logic.

During process startup, the initialization parameters are removed from the `argv` array. After the `IT_Bus::init` invocation completes, the `argv` array contains application parameters only.

It is critical, however, that initialization arguments be entered as flag/value pairs. That is, the `-ORBname` flag must be followed on the command line by the desired value, e.g., `demo.tibrv`.

Controlling Application Logging

Application logging is enable by including the `xmlfile_log_stream` plugin in the `orb_plugins` list. Note that this plugin is included in the default `orb_plugins` list, as shown above. Also note that the `xmlfile_log_stream` plugin is not included in the `orb_plugins` lists within many of the `demo` configuration scopes. If you want to enable logging for these applications, and the applications you develop as described in this document, you will need to include this plugin in your `orb_plugins` list.

Global logging configuration values

To enable usage of the `xmlfile_log_stream` plugin, several other configuration variables must be set. These variable are all set within the default global scope in the `artix.cfg` file.

```
plugins:xmlfile_log_stream:shlib_name =  "it_xmlfile";

plugins:xmlfile_log_stream:filename =  "artix_logfile.xml";
# default: it_bus.log

plugins:xmlfile_log_stream:max_file_size =  "2000000";
# default: 2 mb

plugins:xmlfile_log_stream:use_pid =  "false";
# default: false

# standard logging setting; logs errors and warnings
event_log:filters =  ["*=FATAL+ERROR"];

# very detailed logging
#event_log:filters = ["*=*"];

# transport buffer logging
#event_log:filters =  ["*=FATAL+ERROR+WARNING+INFO_MED"];

# high level informational logging
#event_log:filters =  ["*=FATAL+ERROR+WARNING+INFO_HI"];
```

Using the Logging Functionality

The default configuration settings enable logging only of serious errors and warnings. If you want more exhaustive information, you should either select a different filter list at the default scope, or include a more expansive `event_log:filters` configuration variable within your configuration scope.

If you have trouble running any of the demos, you should enable a high level of logging, which will entail adding the `xmlfile_log_stream` plugin to the `orb_plugins` list and selecting the desired reporting level.

The log will be written into the directory from which the Artix process starts. You can specify the name of the log file through the `plugins:xmlfile_log_stream:filename` configuration variable. The `artix.cfg` file sets the default log file name to `artix_logfile.xml`.

CHAPTER 5

Using the IIOP Transport

In previous chapters, you used Artix™ to implement a C++ Web service that uses SOAP over HTTP. In this chapter, you will reconfigure your application to use IIOP tunneling of SOAP messages as the transport protocol. As you will see, IIOP tunneling simply requires minor changes to the WSDL file; both the client and server code remain unchanged.

In this chapter

This chapter discusses the following topic:

The IIOP Tunneling Demo

The starting point code for this demo is located in the directory `<installationDirectory>\1.3\demos\hello_world\iiop_soap`. This example demonstrates that switching transport protocols from `http` to `iiop` requires only modest reconfiguration within the WSDL file.

The HelloWorld.wsdl File

This file has already been modified. In a text editor, open the file `<installationDirectory>\artix\1.3\demos\hello_world\iiop_soap \client\HelloWorld.wsdl`. Three changes have been made to the file used by the HTTP transport.

Note: The `server` directory contains the same WSDL file.

The iiop Namespace Prefix

In the opening `<definitions>` tag, the namespaces used within the WSDL file are specified. The WSDL file used in the first demo did not have an entry corresponding to the `iiop` transport available within Artix. Consequently, the opening `<definitions>` tag did not include a namespace to be used when specifying information related to the `iiop` transport. To use Artix' `iiop` transport, you need to add another namespace.

The last attribute declaration has been added to the attribute listings within the opening `<definitions>` tag. This attribute defines the namespace prefix `iiop`.

```
<definitions name="HelloWorldService"
    targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://xmlbus.com/HelloWorld" xmlns:xsd=
      "http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://xmlbus.com/HelloWorld/xsd"
    xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
>
    ...
</definitions>
```

There is no harm in including this namespace declaration in all of the WSDL files used by Artix applications.

The <binding> Specification

In this example, the application uses the IIOP transport to send SOAP encoded content. Within the `<binding>` tags the `<soap:binding>` tag specifies the `style` and `transport` used by the binding. Despite the fact that this demo uses the iiop transport, the `transport` attribute specifies http to maintain interoperability with other Web services toolkits.

```
<soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"
```

The <service> Specification

In the original `HelloWorld.wsdl` file, information within the `<service>...</service>` tags specified the `url` at which the Web service could be contacted. When using the `iiop` transport you must replace this entry with information relevant to the `iiop` transport.

The `<service>...</service>` entry has been edited to include the following:

```
<definitions ...>

    <service name="HelloWorldService">
        <port binding="tns:HelloWorldPortBinding"
                name="HelloWorldPort">
            <iiop:address
               location=
                 "corbaloc:iiop:1.2@localhost:55002/tunnel"/>
            <iiop:payload type="octets"/>
            <iiop:policy persistent="true" poaname="MyTunnel"/>
        </port>
    </service>

</definitions>
```

The `iiop` transport related specifications are included in the

```
<iiop:address
      location="corbaloc:iiop:1.2@localhost:55002/tunnel"/>
```

tag. If you are familiar with CORBA's `corbaloc URL` format, you will recognize the format of this specification. In CORBA, the `corbaloc URL` format is used by client processes to obtain an object reference, where `@localhost:55002` refers to the host and port from which the client tries to retrieve the object reference. With Artix, the port on which the server

process should listen for corbaloc requests is specified in the configuration file, which for this installation is the file <installationDirectory>\artix\1.3\etc\domains\artix.cfg.

In a text editor open the configuration file <installationDirectory>\artix\1.3\etc\domains\artix.cfg. Find the tunnel configuration scope, which contains the nested scope demo. Within the tunnel.demo scope, all of the required configuration entries are already included.

The location attribute within the WSDL file is used by client processes and the port number must be the same as the one specified in the configuration file.

Note: Since iiop version 1.2 is the default protocol, the corbaloc URL may be alternatively written as: "corbaloc::localhost:55002/tunnel".

Artix supports other ways to specify the <iiop:address .../> entry. Instead of the corbaloc format, you could use a file url, which provides the path to a file into which Artix will write its object reference. Another alternative is to use the corbaname format, which specifies the name which Artix should use to bind an object reference into a CORBA name service. You will see both of these approaches used in later demos when you learn about Artix/CORBA integration.

The <iiop:payload .../> entry also has alternative values. Use of the type attribute is optional; when it is not present, the value defaults to octets. This specification indicates that the message format specifies the codeset and that Artix does not need to perform codeset negotiation and conversion. The alternative value, string, specifies that Artix is responsible for codeset negotiation and conversion.

Since in this demo the message content is SOAP, which includes a codeset specification, octets is the appropriate value for the type attribute.

Compiling and Running the Application

The `makefile`s include entries that generate the stub, skeleton, and helper classes and build the application's executables.

Compiling the application

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\iiop_soap` directory and issue the command

   ```
   nmake all
   ```

The compilation process creates the `client.exe` and `server.exe` files.

Running the application

You must first start the server process and then run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `artix_env[.bat]` file.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\iiop_soap\server` subdirectory and issue the command:

   ```
   start server -ORBname tunnel.demo
   ```

   Note the use of the command line arguments `-ORBname tunnel.demo`, which causes the server process to start under the similarly named configuration scope.
3. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\iiop_soap\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows.

Terminating the server process

Issue the `Ctrl-C` command in the corresponding command window.

CHAPTER 6

Using the Tuxedo Transport

In this chapter, you will reconfigure your application to use Tuxedo as the transport protocol. As you will see, this process simply requires minor changes to the WSDL file and redefinition of the Tuxedo runtime environment. Additionally, small changes to the client and server code are required.

In this chapter

This chapter discusses the following topics:

The Demo Code

The starting point code for this demo is located in the `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp` directory. The source code files, starting point configuration files, and WSDL file have been placed into the appropriate directories.

The Client and Server Source Code

This code is basically unchanged from earlier demos with one exception. In both the client and server processes, initialization of the runtime environment occurs during execution of the method

```
IT_Bus::init(argc, argv);
```

Within this method, an ORB is initialized. In the demo that utilized the HTTP transport, this ORB was configured under the default scope within the configuration domain `artix`. In the demo that utilized the IIOP_tunnel transport, a different configuration scope, which included entries relevant to the underlying iiop_tunneling functionality, was specified through the `-ORBname` command line parameter.

For this demo, it is important to configure the ORB under a scope that adds the `tuxedo` plugin to the `orb_plugins` listing. To accomplish this, the overloaded `init` method is used in both the client and server applications. In the client application the appropriate configuraton scope is `demo.tuxedo`.

```
IT_Bus::init(argc, argv, “demo.tuxedo”);
```

For the server application use the scope `demo.tuxedo.server`.

```
IT_Bus::init(argc, argv, “demo.tuxedo.server”);
```

Alternatively, you could use the `-ORBname` command line parameter when starting each process.

The demo.tuxedo scope

The directory `<installationDirectory>\artix\1.3\etc\domains` includes the configuration file `artix.cfg`. Open this file in a text editor and find the `demo` scope. Note that the `demo` scope contains multiple nested scopes. You will add two nested scopes– `tuxedo` and `tuxedo.demo`–under `demo`.

Edit the Artix configuration file

Directly under the opening brace of the `demo` scope, add the nested `tuxedo` configuration scope.

```
demo
{
 tuxedo {
   orb_plugins=["xmlfile_log_stream", "soap", "tuxedo"];
   event_log:filters=["*=FATAL+ERROR"];
   server {
     plugins:tuxedo:server="true";
   };
 };

};
```

The HelloWorld.wsdl File

This file has already been modified. In a text editor, open the file `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp\client\HelloWorld.wsdl`. Three changes have been made to the WSDL file used in the earlier examples.

The tuxedo Namespace Prefix

In the opening `<definitions>` tag, the namespaces used within the WSDL file are specified. Previously used WSDL files obviously had no knowledge of the Tuxedo transport support available within Artix. Consequently, the opening `<definitions>` tag did not include a namespace to be used when specifying information related to the Tuxedo transport. To use Artix' Tuxedo transport, you need to add another namespace.

The last attribute declaration has been added to the attribute listings within the opening `<definitions>` tag. This attribute defines the namespace prefix `tuxedo`.

```
<definitions name="HelloWorldService"
    targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://xmlbus.com/HelloWorld" xmlns:xsd=
      "http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://xmlbus.com/HelloWorld/xsd"
    xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
    xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
>
    ...
</definitions>
```

There is no harm in including this namespace declaration in all of the WSDL files used by Artix applications. In fact, note that this WSDL file also includes the iiop namespace prefix declaration from the IIOP_tunnel example.

The <binding> Specification

In this example, the application uses the Tuxedo transport to send SOAP encoded content. Within the `<binding>` tags the `<soap:binding>` tag specifies the `style` and `transport` used by the binding. Despite the fact that this demo uses the Tuxedo transport, the `transport` attribute specifies http to maintain interoperability with other Web services toolkits.

```
<soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"
```

The <service> Specification

In the original `HelloWorld.wsdl` file, information within the `<service>…</service>` tags specified the `url` at which the Web service could be contacted. When using the Tuxedo transport you must replace this entry with information relevant to the Tuxedo transport.

The `<service>…</service>` entry has been edited to include the following:

```
<definitions ...>

    <service name="HelloWorldService">
        <port binding="tns:HelloWorldPortBinding"
                name="HelloWorldPort">
           <tuxedo:server>
             <tuxedo:service name="HelloWorld"/>
           </tuxedo:server>
        </port>
    </service>

</definitions>
```

The Tuxedo transport related specifications are included in the

```
<tuxedo:server>
   <tuxedo:service name="HelloWorld"/>
</tuxedo:server>
```

tag, which specifies the Tuxedo service name under which your server application will run.

The Tuxedo Configuration

You will need to set environment variables and generate an application specific configuration file for the Tuxedo application.

Edit the setenv.cmd File

This file sets the Tuxedo related environment variables.

Open the file `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp\setenv.cmd` in a text editor and confirm that the `TUXDIR` and `APPDIR` entries are correct.

Edit the ubbhelloWorld Configuration File

Open the file `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp\ubbhelloWorld` in a text editor. You must replace all of the entries within the brackets (<...>) with valid entries.

With the exception of the machine name entry, it's important that you accept the other suggested entries (modifying `PATH` values as appropriate). While you are generally free to give these entries other values, there must be consistency between these entries and the commands used to compile and run the Tuxedo application. If you do not honor the suggested content, you will have difficulty with subsequent steps.

Configuring, Compiling and Running the Application

All of the source code and configuration files are in their appropriate directories. You must first generate the Tuxedo binary configuration file and then compile the C++ application.

Generating the Application Specific Tuxedo Binary Configuration File

This file includes the information specified in the `ubbhelloWorld` configuration file.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `artix_env[.bat]` file.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp` directory and run the file `setenv[.cmd]`.
3. Issue the command

   ```
   tmloadcf ubbhelloWorld
   ```

   Enter "`y` " and press the return/enter key to confirm that you want to create/recreate the binary configuration file.

Compiling the Application Code

The `makefile`s include entries that incorporate the copied files into your executable.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp` directory and run the file `setenv[.cmd]`.

   Alternatively, you can continue to use the command window from the section Generating the Application Specific Tuxedo Binary Configuration File.
3. While in the `tux_tp` directory, issue the command

   ```
   nmake all
   ```

The compilation process creates the `client.exe` and `server.exe` files in their respective directories.

Running the Application

You must first start the Tuxedo runtime, which starts your server process. Then you can run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `artix_env[.bat]` file.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp` directory and run the file `setenv[.cmd]`.

   Alternatively, you can continue to use the command window from steps Generating the Application Specific Tuxedo Binary Configuration File or Compiling the Application Code.
3. While in the `tux_tp` directory, start the Tuxedo server process with the command

   ```
   tmboot -y
   ```
4. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\tux_tp\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the client command window.

Stopping the Tuxedo server process

1. Stop the process by issuing the command

   ```
   tmshutdown -y
   ```

   in the command window.

CHAPTER 7

Using the WebSphere™ MQ Transport

In earlier chapters you used Artix™ to implement a C++ Web service that sent SOAP over the HTTP, IIOP, and Tuxedo. In this example, you will reconfigure your application to use WebSphere MQ as the transport protocol. As you will see, this process simply requires minor changes to the WSDL file and creation of application specific queues.

In this chapter

This chapter discusses the following topics:

Creating the WebSphere MQ Queues

The WebSphere MQ installation process deploys the product as a Windows service, which starts automatically when you boot your computer. You may have reconfigured this service for manual startup. Consequently, you will need to start both the WebSphere MQ Service before you can create your application's queues.

Starting the WebSphere MQ Service

You can start the WebSphere MQ Service from either a task bar icon or from the Windows control panel Services window.

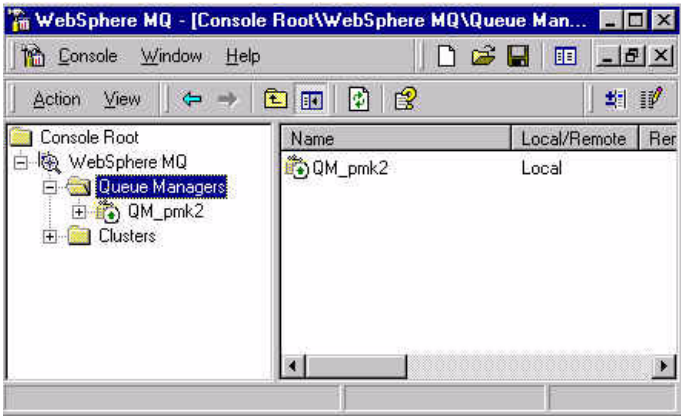
From the task bar, right click on the icon and select Start WebSphere MQ from the popup menu. The task bar icon's color changes from red to blue during the startup process. When the services are fully operational, the icon's color changes to green.

From the Services window, highlight the IBM MQSeries entry and click on the Start command button or menu selection. Again, the task bar icon's color changes from red to blue during the startup process. When the services are fully operational, the icon's color changes to green. Close the Services window.

Open the WebSphere MQ Explorer

From the Start menu, select the WebSphere MQ Explorer entry.

This opens the explorer.



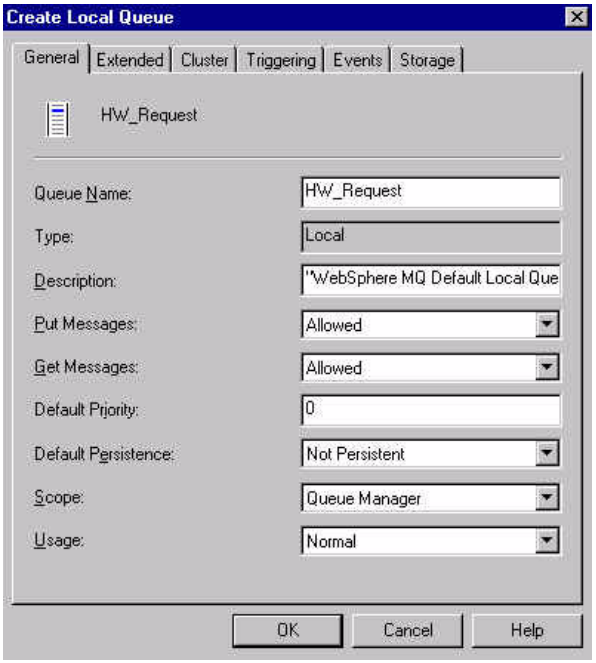
Note that the default Queue Manager (`QM_<hostName>`) is already running. Highlight the Queue Managers icon and either right click or click on the Action menu bar button. Select the New > Queue Manager menu item and create the queue manager `MY_DEF_QM`.

Note: This queue manager is also used by the `switch` demo. If you have already run the `switch` demo, you will have already created the queue manager.

Creating the HW_REQUEST and HW_REPLY Queues

Your application requires two queues; the client will put requests onto the request queue and the server will put responses onto the response queue. It does not matter what you name these queues. You may also create additional queue managers and possibly assign each queue to a different manager. In this demo, you will simply create two queues and assign them to the MY_DEF_QM queue manager.

Right click on the Queue icon under the MY_DEF_QM queue manager icon, and select New > Local Queue from the popup menu. Alternatively, click on the Action command button and select New > Local Queue from the drop down menu. This opens the Create Local Queue window. You only need to enter a name for the queue, click on the OK command button, and then, in the WebSphere MQ message window, click on the Don't Share in Cluster command button.



Create two queues named: HW_REQUEST and HW_REPLY and close the explorer window.

The Demo Code

The starting point code for this demo is located in the `<installationDirectory\artix\1.3\demos\hello_world\mq_soap` directory. The source code files, starting point configuration files, and WSDL file have been placed into the appropriate directories.

The client and server source code

This code is unchanged from the earlier demos involving SOAP over HTTP and SOAP over IIOP.

The HelloWorld.wsdl File

This file has already been modified. In a text editor, open the file `<installationDirectory\artix\1.3\demos\hello_world\mq_soap \client\HelloWorld.wsdl`. Three changes have been made to the WSDL file used in the earlier examples.

The mq namespace prefix

In the opening `<definitions>` tag, the namespaces used within the WSDL file are specified. The WSDL files used previously had no knowledge of the WebSphere MQ transport support available within Artix. Consequently, the opening `<definitions>` tag did not include a namespace to be used when specifying information related to the WebSphere MQ transport. To use the WebSphere MQ transport, you need to add another namespace.

The last attribute declaration has been added to the attribute listings within the opening <definitions> tag. This attribute defines the namespace prefix mq.

```
<definitions name="HelloWorldService"
    targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://xmlbus.com/HelloWorld" xmlns:xsd=
      "http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://xmlbus.com/HelloWorld/xsd"
    xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
    xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
    xmlns:mq="http://schemas.iona.com/transports/mq"
>
    ...

</definitions>
```

There is no harm in including this namespace declaration in all of the WSDL files used by Artix applications. In fact, note that this WSDL file also includes the iiop namespace prefix declaration and the Tuxedo namespace prefix declaration from the earlier examples.

The <binding> Specification

In this example, the application uses the WebSphere MQ transport to send SOAP encoded content. Within the <binding> tags the <soap:binding> tag specifies the style and transport used by the binding. Despite the fact that this demo uses the WebSphere MQ transport, the transport attribute specifies http to maintain interoperability with other Web services toolkits.

```
<soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"
```

The <service> specification

When using the WebSphere MQ transport you must replace information within the <service>…</service> tags with information relevant to the WebSphere MQ transport.

The `<service>…</service>` entry has been edited to include the following:

```
<definitions ...>
  <service name="HelloWorldService">
    <port binding="tns:HelloWorldPortBinding"
          name="HelloWorldPort">
      <mq:client  QueueManager="MY_DEF_QM"
                        QueueName="HW_REQUEST"
                        AccessMode="send"
                        ReplyQueueManager="MY_DEF_QM"
                        ReplyQueueName="HW_REPLY"
        />

        <mq:server  QueueManager="MY_DEF_QM"
                          QueueName="HW_REQUEST"
                          ReplyQueueManager="MY_DEF_QM"
                          ReplyQueueName="HW_REPLY"
                          AccessMode="receive"
        />
    </port>
  </service>>
</definitions>
```

The WebSphere MQ transport related specifications are included in the

```
<mq:client …/>
```

and

```
<mq:server …/>
```

tags.

The artix.cfg File

The configuration file – `artix.cfg` – is located in the `<installationDirectory>\artix\1.3\etc\domains` directory. The `orb_plugins` entry is near the beginning of this file. This configuration entry does not include the `mq` plugin, which is required to run this demo.

In a text editor, open the `artix.cfg` file and find the `orb_plugins` entry. Add the `mq` plugin to the listing.

```
orb_plugins=["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop", "soap", "http", "tunnel", "ws_orb", "fixed", "mq"];
```

Configuring, Compiling and Running the Application

All of the source code and configuration files are in their appropriate directories.

Compiling the Application Code

The `makefiles` include entries that generate the stub, skeleton, and helper classes and create your executable.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory.
2. Run the batch file `artix_env[.bat]`.
3. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\mq_soap` directory and issue the command

   ```
   nmake all
   ```

The compilation process creates the `client.exe` and `server.exe` files in their respective directories.

Running the Application

Be certain that your WebSphere MQ Services and queue manager are running.

1. Open a command window to the `<installationDirectory>\artix\6.0\bin` directory and run the `artix_env[.bat]` file.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\mq_soap\srv` directory and issue the command

   ```
   start server
   ```
3. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\mq_soap\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the client command window.

Terminating the server process	Issue the `Ctrl-C` command in the corresponding command window.
Stopping the queue manager	If desired, you can stop and restart the queue manager from the WebSphere MQ Explorer window. Open the explorer and highlight the icon corresponding to your queue manager. Right click, or click on the Action command button, and select the appropriate action from the menu.
Stopping the WebSphere MQ services	You can stop the WebSphere MQ Service from either a task bar icon or from the Windows control panel Services window. From the task bar, right click on the icon and select Stop WebSphere MQ from the popup menu; confirm your action in the message box. The task bar icon's color changes from green to blue during the shutdown process. When shutdown has completed, the icon's color changes to red. From the Services window, highlight the IBM MQSeries entry and click on the Stop command button or menu selection; confirm your action in the message box. Close the Services window.

Further Considerations

This example assumes there is one client process sending requests through WebSphere MQ to a single server process. In a more realistic scenario, there would be multiple clients issuing requests. Since all of the clients would be using the same request and reply queues, it is possible that one client might retrieve responses meant for another client. This problem can be easily managed by specifying the `CorrelationStyle` attribute with the WebSphere MQ port information.

The `CorrelationStyle` attribute specifies a mechanism that the processes and queue manager will use to uniquely identify messages and their corresponding clients. If you edit the WSDL file to include the `CorrelationStyle` attribute, responses will be returned to the client issuing the corresponding request.

The following fragment illustrates how to add this information to the WSDL file. Refer to the Artix product documentation for a more complete discussion of this, and other, attributes.

```
<definitions ...>
  <service name="HelloWorldService">
    <port binding="tns:HelloWorldPortBinding"
          name="HelloWorldPort">
      <mq:client  QueueManager="MY_DEF_QM"
                          QueueName="HW_REQUEST"
                          AccessMode="send"
                          ReplyQueueManager="MY_DEF_QM"
                          ReplyQueueName="HW_REPLY"
                          CorrelationStyle="messageId"
        />

        <mq:server  QueueManager="MY_DEF_QM"
                           QueueName="HW_REQUEST"
                           ReplyQueueManager="MY_DEF_QM"
                           ReplyQueueName="HW_REPLY"
                           AccessMode="receive"
                           CorrelationStyle="messageId"
        />
    </port>
  </service>>
</definitions>
```

CHAPTER 8

Using the TIBCO Rendezvous™ Transport

As with the other transport protocols, adapting your HelloWorld application to the TIBCO Rendezvous transport primarily involves changes to the WSDL file. Unlike the WebSphere™ MQ transport, there is no need to create "subjects" or "queues" as part of the reconfiguration. The only change you must make to your application code is to insure that the Artix TIBCO Rendezvous plugin is loaded during initialization of the client and server applications.

In this chapter

This chapter discusses the following topics:

The Demo Code

The starting point code for this demo is located in the `<installationDirectory>\artix\1.3\demos\hello_world\tibrv` directory. The source code files, starting point configuration files, and WSDL file have been placed into the appropriate directories.

Both the client application and the server process become "Artix-aware" during the execution of the `IT_Bus::init` invocation. This method loads the underlying Artix runtime environment into the running process. The runtime services that each application uses are implemented through libraries that are loaded during this initialization.

The Artix configuration file – `artix.cfg` – includes all of the information needed to start an Artix process under a default configuration. This default configuration does not, however, include the library that provides access to the TIBCO Rendezvous transport; you must use the alternative version of the `IT_Bus::init` method, specifying the `demo.tibrv` scope, which adds the `tibrv` plugin to the `orb_plugins` listing. As a result, the signature of the overloaded `init` method used in both the client application and server process is:

```
IT_Bus::init(argc, argv, "demo.tibrv");
```

Both the `client.cxx` and `server.cxx` files have been edited to include this modification.

The artix.cfg File

The configuration file – `artix.cfg` – is located in the `<installationDirectory>\artix\1.3\etc\domains` directory. Within this file, the `demo.tibrv` scope redefines the `orb_plugins` variable, including the `tibrv` plugin in the list.

The HelloWorld.wsdl File

This file has already been modified. In a text editor, open the file `<installationDirectory>\artix\1.3\demos\hello_world\tibrv\client\HelloWorld.wsdl`. This file differs from the WSDL file used in the earlier demos in three sections: an additional namespace prefix is defined; the `<binding>` specification includes entries specific to the TIBCO Rendezvous transport; and the `<service>` specification includes entries needed to contact the transport.

The tibrv namespace prefix

In the opening `<definitions>` tag, the namespaces used within the WSDL file are specified. To use the TIBCO Rendezvous transport, you need to add another namespace.

The last attribute declaration has been added to the attribute listings within the opening `<definitions>` tag. This attribute defines the namespace prefix `tibrv`.

```
<definitions name="HelloWorldService"
    targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://xmlbus.com/HelloWorld" xmlns:xsd=
      "http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://xmlbus.com/HelloWorld/xsd"
    xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
    xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
    xmlns:mq="http://schemas.iona.com/transports/mq"
    xmlns:tibrv="http://schemas.iona.com/transports/tibrv"
>
    ...
</definitions>
```

The <binding> Specification

This section now includes a description of each of the operations in a format that has been developed to integrate Artix applications with the TIBCO Rendezvous protocol.

```
<binding name="HelloWorldPortBinding"
          type="tns:HelloWorldPortType">
    <tibrv:binding/>
    <operation name="greetMe">
        <tibrv:operation/>
        <input name="greetMe">
            <tibrv:input/>
        </input>
        <output name="greetMeResponse">
            <tibrv:output/>
        </output>
    </operation>
    <operation name="sayHi">
        <tibrv:operation/>
        <input name="sayHi">
            <tibrv:input/>
        </input>
        <output name="sayHiResponse">
            <tibrv:output/>
        </output>
    </operation>
</binding>
```

Obviously this section includes the same operations as the `<binding>` specification in previous WSDL files, but the namespace associated with the operation signatures is `tibrv` rather than `soap`.

The <service> specification

The `<service>` specification now includes information that the Artix runtime environment needs to interact with the TIBCO Rendezvous transport.

The <service>…</service> entry has been edited to include the following:

```
<service name="HelloWorldService">
    <port binding="tns:HelloWorldPortBinding"
          name="HelloWorldPort">
        <tibrv:port
           subject="Artix.HelloWorld"
        />
    </port>
</service>
```

Within the application, this is the only information specific to the TIBCO Rendezvous transport. There are no Rendezvous specific configuration information or coding within the application files that you provide.

Compiling and Running the Application

All of the source code and configuration files are in their appropriate directories.

Compiling the Application Code

The `makefile`s include entries that incorporate the files into your executable.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\hello_world\tibrv` directory and issue the command

   ```
   nmake all
   ```

The compilation process creates the `client.exe` and `server.exe` files in their respective directories.

Running the Application

There is no need to start the TIBCO Rendezvous routing daemon before running this application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `artix_env[.bat]` file.
2. Move to the `<installationDirectory>\artix\1.3\hello_world\tibrv\server` directory and issue the command

   ```
   start server
   ```
3. Move to the `<installationDirectory>\artix\1.3\hello_world\tibrv\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the client and server command windows.

Terminating the server process

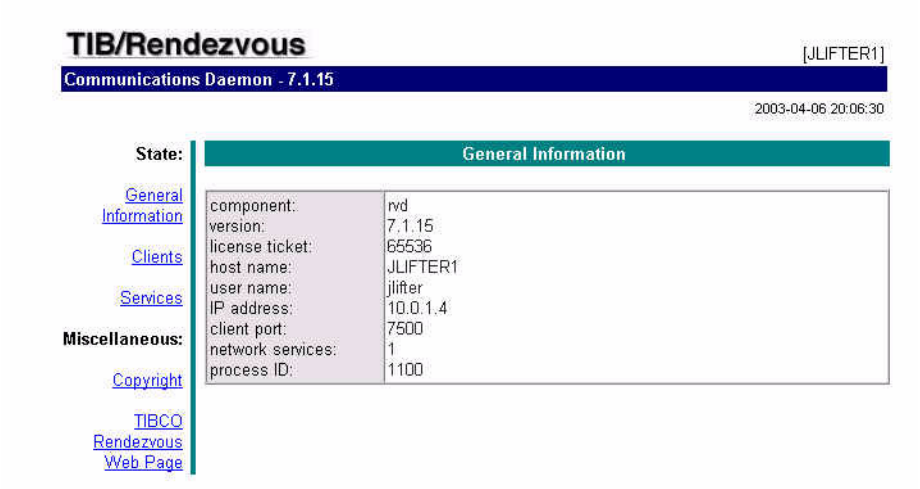
Issue the `Ctrl-C` command in the corresponding command window.

Note: Do not stop the server until you complete the monitoring steps described in the following section.

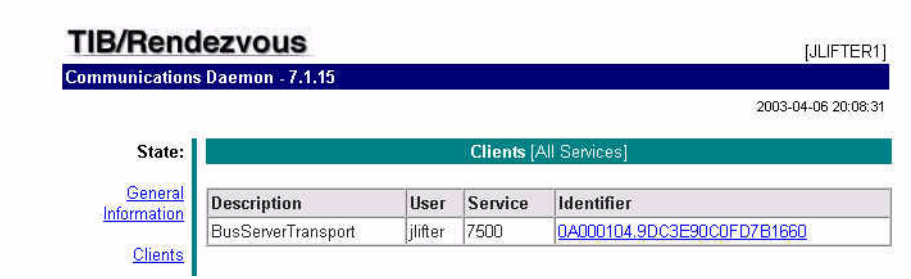
Monitoring the TIBCO Rendezvous Environment

The TIBCO Rendezvous routing daemon will not start until your server process runs and the Artix runtime accesses Rendezvous.

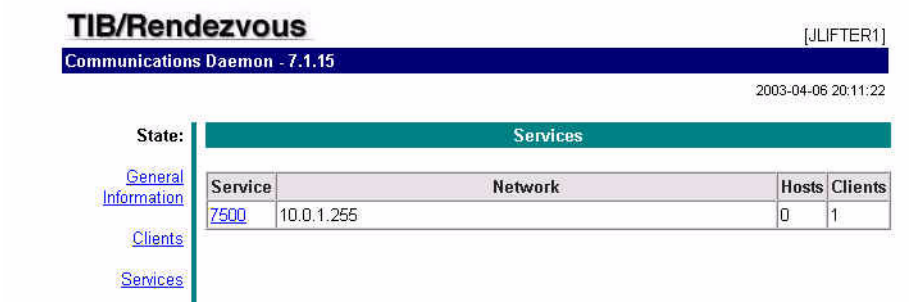
After starting the server process, enter the URL `http://<hostname>:7850` into your browser. The initial screen provides general information about your Rendezvous environment.



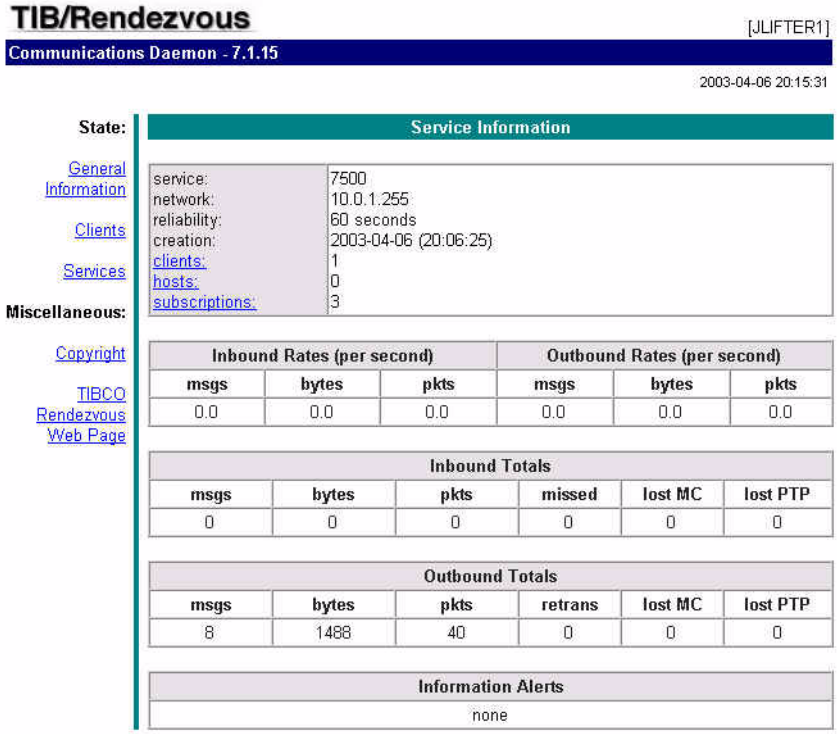
Click on the Clients hyperlink and the Clients (All Services) window confirms that your server process is running as a TIBCO Rendezvous client.



Next, click on the Services hyperlink and the Services window includes the default TIBCO Rendezvous routing daemon in the listing of active services.



Finally, click on the port number hyperlink (`7500`). This opens the Service Information window in which you can monitor the messages passing through the routing daemon.



Now, run your client program several times and observe the updated service information.

CHAPTER 9

Using CORBA Applications and Transport

In this chapter, you will learn how to integrate existing CORBA applications with Artix™. You will study a demo in which a CORBA client sends CORBA requests to an Artix server; the server process then delivers the requests to a C++ object. You must now edit the WSDL file to include the particulars of the CORBA data types and to specify how the Artix process should publish an object reference. Invocations sent to the Artix server process are reformatted as C++ method calls against the Artix implementation object. In a second example, you will deploy an Artix client against the Artix server; communication between the client and server processes uses CORBA invocations over IIOP. Finally, you will deploy a CORBA server, which will be used by your Artix client; again, communication between the client and server processes will use CORBA invocations over IIOP.

In this chapter

This chapter discusses the following topics:

The CORBA Client—Artix Server Demo

The starting point code for this demo is located in the `<installationDirectory>\artix\1.3\demos\corba\corba_client_artix_server` directory. The source code and IDL file are complete. The WSDL file, which is used by all the demos described in this chapter, is created within the `<installationDirectory>\artix\1.3\demos\corba\common` directory.

The HelloWorld.idl File

The IDL file is only used by the client application, which is a CORBA application. For this application, the IDL file contains only a single interface definition within a single module.

```
module HW {
  interface HelloWorld {
    string sayHi ();
    string greetMe (in string user);
  };
};
```

When you create the WSDL file, you must provide data type definitions and message declarations that correspond to these CORBA operations and parameters.

The HelloWorld.wsdl File

Unlike the earlier demos, which used a prewritten WSDL file, this demo uses IONA's IDL compiler to generate a WSDL file directly from the IDL file.

Generating the HelloWorld.wsdl file

IONA's IDL compiler responds to several command line flags that specify how to process an IDL file into a WSDL file. This is the same IDL compiler used to produce CORBA stubs, skeletons, and starting point servant code. By using the appropriate command line flags, the compiler produces a WSDL file instead of the CORBA classes.

You can use four flags to control generation of the WSDL file.

- `-wsdl`
  The flag that directs the IDL compiler to produce a WSDL file. This is the only required flag and it must be followed by the name of the IDL file.

- `-a<address>`
  The flag that specifies an absolute address through which the object reference may be accessed. The `<address>` may be a relative or absolute path to a file, or a `corbaname` URL. There is no white space between the `-a` and `<address>` entries.
- `-f<file>`
  The flag that specifies a file containing a string representation of an object reference. The contents of this file will be incorporated into the WSDL file. The `<file>` must exist when you run the IDL compiler. There is no white space between the `-f` and `<file>` entries.
- `-O<dir>`
  The flag used to specify the directory into which the WSDL file should be written. There is no white space between the `-O` and `<dir>` entries.

To combine multiple flags in the same command, use a colon (":") delimited list. Note that the colon is only interpreted as a delimiter if it is followed by a dash ("- "). Consequently, the colons in a `corbaname` URL are interpreted as part of the URL syntax and not as delimiters.

To create the WSDL file:

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and issue the command

   ```
   artix_env[.bat]
   ```

2. Move to the `<installationDirectory>\artix\1.3\demos\corba\common` directory and issue the command

   ```
   idl -wsdl:-afile://../../common/HelloWorld.ior
        HelloWorld.idl
   ```

   This generates the `HelloWorld.wsdl` file that includes a direct reference to the file into which the server process will write an object reference.

3. Alternatively, issue the command

   ```
   idl -wsdl:-acorbaname:rir:/NameService#helloWorld
        HelloWorld.idl
   ```

   This generates the `HelloWorld.wsdl` file that includes a `corbaname` URL. An object reference is bound under the name `helloWorld`. Since

the object reference is bound at the root level of the name service, you do not need to create a name context to run this demo.

Note: Be careful regarding capitalization. The command line flag entries are case sensitive even on Windows. Capitalization in your generated WSDL file must match the capitalization used in the prewritten code. The name of the IDL file is `HelloWorld.idl`.

The namespace prefixes

In the opening `<definitions>` tag, the namespaces used within the WSDL file are specified.

Two attribute declarations have been added to the attribute listings within the opening `<definitions>` tag. These attributes define the namespace prefixes `corba` and `corbatm`.

```
<definitions name="HelloWorld.idl"
 targetNamespace="http://schemas.iona.com/idl/HelloWorld.idl"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://schemas.iona.com/idl/HelloWorld.idl"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://schemas.iona.com/idltypes/HelloWorld.idl"
 xmlns:corba="http://schemas.iona.com/bindings/corba"
 xmlns:corbatm="http://schemas.iona.com/typemap
    /corba/HelloWorld.idl"
>
    ...
</definitions>
```

Note that the `xsd1` prefix has been associated with the IDL file used to generate the WSDL file.

The <types> Specification

In the earlier demos, since there were no complex or derived types that needed further description, this section of the WSDL file was not used. When using the CORBA transport, you must use this section to define each of the types that are used by the application's methods. For your HelloWorld

application, you must define the return values from the `sayHi` and `greetMe` messages as well as the parameter to the `greetMe` message, as shown in the following extract.

```
<types>
  <schema targetNamespace=
       "http://schemas.iona.com/idltypes/HelloWorld.idl"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <element name="HW.HelloWorld.sayHi.return"
   type="xsd:string"/>
      <element name="HW.HelloWorld.greetMe.user"
   type="xsd:string"/>
      <element name="HW.HelloWorld.greetMe.return"
   type="xsd:string"/>
      <element name="HW.HelloWorld.greetAny.msg"
   type="xsd:anyType"/>
      <element name="HW.HelloWorld.greetAny.return"
   type="xsd:anyType"/>
  </schema>
</types>
```

The <message> specification

As in the earlier WSDL files, you must describe the messages. These descriptions use the `element` attribute, which gives the fully qualified name for each part of the message, as shown in the following extract.

```
<message name="HW.HelloWorld.sayHi"/>
<message name="HW.HelloWorld.sayHiResponse">
    <part name="return"
            element="xsd1:HW.HelloWorld.sayHi.return"/>
</message>
<message name="HW.HelloWorld.greetMe">
    <part name="user"
            element="xsd1:HW.HelloWorld.greetMe.user"/>
</message>
<message name="HW.HelloWorld.greetMeResponse">
    <part name="return"
            element="xsd1:HW.HelloWorld.greetMe.return"/>
</message>
```

```
<message name="HW.HelloWorld.greetAny">
    <part name="msg"
            element="xsd1:HW.HelloWorld.greetAny.msg"/>
</message>
<message name="HW.HelloWorld.greetAnyResponse">
    <part name="return"
            element="xsd1:HW.HelloWorld.greetAny.return"/>
</message>
```

The <portType> Specification

The port type information corresponds exactly to the information provided in the other WSDL files.

```
<portType name="HW.HelloWorld">
  <operation name="sayHi">
    <input message="tns:HW.HelloWorld.sayHi"
             name="sayHi"/>
    <output message="tns:HW.HelloWorld.sayHiResponse"
             name="sayHiResponse"/>
  </operation>
  <operation name="greetMe">
    <input message="tns:HW.HelloWorld.greetMe"
             name="greetMe"/>
    <output message="tns:HW.HelloWorld.greetMeResponse"
             name="greetMeResponse"/>
  </operation>
  <operation name="greetAny">
    <input message="tns:HW.HelloWorld.greetAny"
             name="greetAny"/>
    <output message="tns:HW.HelloWorld.greetAnyResponse"
             name="greetAnyResponse"/>
  </operation>
</portType>
```

The <binding> specification

This section differs significantly from your earlier WSDL files, which were associated with the `soap` namespace. The `corba` namespace is used in this application. Because there are no complex or derived types in this application, the `typeMapping` section does not include any entries, but a more involved application would have additional content within this section.

```
<binding name="HW.HelloWorldBinding" type="tns:HW.HelloWorld">
  <corba:binding
            repositoryID="IDL:HW/HelloWorld:1.0"/>
  <operation name="sayHi">
    <corba:operation name="sayHi">
      <corba:return name="return"
             idltype="corba:string"/>
    </corba:operation>
    <input name="sayHi"/>
    <output name="sayHiResponse"/>
  </operation>
 <operation name="greetMe">
    <corba:operation name="greetMe">
      <corba:param name="user" mode="in"
              idltype="corba:string"/>
      <corba:return name="return"
              idltype="corba:string"/>
    </corba:operation>
    <input name="greetMe"/>
    <output name="greetMeResponse"/>
  </operation>
  <operation name="greetAny">
    <corba:operation name="greetAny">
      <corba:param name="msg" mode="in"
              idltype="corba:any"/>
      <corba:return name="return"
              idltype="corba:any"/>
    </corba:operation>
    <input name="greetAny"/>
    <output name="greetAnyResponse"/>
  </operation>
</binding>
```

Note the value of the `<corba:binding repositoryID=…>` entry. The value you provide here becomes the type ID embedded in the object reference. This value must match the type ID that would be created from the interface definition by the IDL compiler. The IDL file used by the client application is

expecting an object reference of type `IDL:HW/HelloWorld:1.0`, where `HelloWorld` is the interface name and `HW` the module enclosing this interface definition.

The <service> specification

The content of the `<service>` and `<port>` tags is similar to your earlier WSDL files, it is the content of the `<address>` tag that specifies use of the CORBA transport.

```
<service name="HW.HelloWorldService">
  <port name="HW.HelloWorldPort"
          binding="tns:HW.HelloWorldBinding">
    <corba:address
          location="file://../../common/HelloWorld.ior"/>
  </port>
</service>
```

Note the contents of the `<corba:address>` tag. The `location` attribute specifies that the Artix process should write a string representation of its interoperable object reference to the file `HelloWorld.ior`. This file will be written into the `common` directory so that it will be easily accessible by the client application. Consequently, the CORBA client code must be able to read this file and convert the string into an object reference. In this demo, the location of the `HelloWorld.ior` file is hard-coded into the CORBA client code.

If you installed the Artix product on top of (or with access to) an Orbix installation, you can use the CORBA Name Service to hold the object bindings instead of local files. In this situation, you would edit the `<service>` specification so that the `location` attribute's value is a `corbaname` URL.

```
<corba:address
       location="corbaname:rir:/NameService#helloWorld"/>
```

Since the client code is hard-coded to source the object reference from a file, you also need to edit this code.

Find the line of code:

```
tobj = orb->string_to_object(objref_string.in());
```

and change to:

```
tobj = orb->string_to_object
       (“corbaname:rir:/NameService#helloWorld”);
```

Note: Since you did not comment out the client code that reads the IOR from a file, be certain not to try this change until after you have successfully run the application using the file approach.

Compiling and Running the Application

Since the coding is completed, you can simply compile and run the application.

Compiling the Application Code

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\corba\corba_client_artix_server` directory and issue the command

   ```
   nmake all
   ```

Running the Application

You must first start the server process and then run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\corba\corba_client_artix_server\server` subdirectory and issue the command:

   ```
   start server
   ```

   Note that the `HelloWorld.ior` file is written into the `common` directory.
3. Move to the `<installationDirectory>\artix\1.3\demos\corba\corba_client_artix_server\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows.

Using the CORBA Name Service

If you want to run this application using the Orbix CORBA Name Service, you should:

1. Stop the server process.

2. Modify the `artix.cfg` configuration file, which is in the directory `<installationDirectory>\artix\1.3\etc\domains`. You must add the following three lines within the global scope; you may place these entries at the top of the file.

```
initial_references:NameService:reference=
  “corbaloc::localhost:3075/NameService”;
url_resolvers:corbaname:plugin=“naming_resolver”;
plugins:naming_resolver:shlib_name=“it_naming”;
```

   Where `3075` is the port assigned to your Orbix Locator Daemon. You may also substitute a machine name or IP address for the entry `localhost` and a different `port` number, if appropriate.
3. Edit the `HelloWorld.wsdl` and `client.cxx` files as described in the discussion of the `<service>` specification.
4. Recompile and run the application as described earlier in this section.

Terminating the server process

Issue the `Ctrl-C` command in the corresponding command window.

The Artix Client—Artix Server Demo

The starting point code for this demo is located in the `<installationDirectory>\artix\1.3\demos\corba\artix_client_artix_server` directory. The WSDL file is located in the `<installationDirectory>\artix\1.3\demos\corba\common` directory. Additionally, this demo uses the same server process as the CORBA Client—Artix Server demo in the previous section. Consequently, the client executable is built using the stub and helper classes originally generated into the `<installationDirectory>\artix\1.3\demos\corba\corba_client_artix_server\server` directory.

Compiling and Running the Application

Since the coding is completed, you can simply compile and run the application.

Compiling the Application Code

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\corba\artix_client_artix_server` directory and issue the command

   ```
   nmake all
   ```

Running the Application

You must first start the server process and then run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `batch file artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\corba\corba_client_artix_server\server` subdirectory and issue the command:

   ```
   start server
   ```

   Note that the `HelloWorld.ior` file is written into the `common` directory.
3. Move to the `<installationDirectory>\artix\1.3\demos\corba\artix_client_artix_server\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows. To convince yourself that the Artix client application is using the CORBA object reference to invoke on the Artix server, change the name of the `HelloWorld.ior` file and try rerun the client.

Terminating the server process

Issue the `Ctrl-C` command in the corresponding command window.

The Artix Client—CORBA Server Demo

The starting point code for this demo is located in the `<installationDirectory>\artix\1.3\demos\corba\artix_client_corba_server` directory. This demo uses the same client process as the Artix Client—Artix Server demo, so the only code in this demo relates to the CORBA server process.

Compiling and Running the Application

Since the coding is completed, you can simply compile and run the application.

Compiling the Application Code

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\corba\artix_client_corba_server` directory and issue the command

   ```
   nmake all
   ```

Running the CORBA Server Process

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\corba\artix_client_corba_server\server` subdirectory and issue the command:

   ```
   start server
   ```

The CORBA server process starts in a new command window and the `HelloWorld.ior` file is written into the `common` directory.

Testing the CORBA server

Between the client code included in the CORBA Client—Artix Server demo and the server code from this Artix Client—CORBA Server demo, you have a complete CORBA application. You will use the CORBA client application to confirm that the CORBA server runs as anticipated.

Running the CORBA Client Process

1. From the command window used in "Running the CORBA Server Process" on page 92, move to the `<installationDirectory>\artix\1.3\demos\corba\corba_client_artix_server\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows.

Running the Artix Client Process

1. From the command window used in "Running the CORBA Server Process" on page 92, move to the `<installationDirectory>\artix\1.3\demos\corba\artix_client_artix_server\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

2. Observe the messages in both the server and client command windows.

Terminating the server process

Issue the `Ctrl-C` command in the corresponding command window.

CHAPTER 10

Routing

In previous chapters, you learned the basics of writing an Artix™ client and server process and how to use the WSDL file to select a transport protocol. In this chapter you will learn how to use the WSDL file to create a message routing, that is, redirect an invocation using a different transport protocol and to route requests for specific operations to different server processes. You will also learn how to route requests to different server processes based on the content of the request. Content based routing differs from protocol and operation based routing in that the routing logic is provided through your coding rather than through the information within the WSDL file.

In this chapter

This chapter discusses the following topics:

The Routing Demos

The starting point code for this demo is located in the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing` and `<installationDirectory>\artix\1.3\demos\routing\operation_routing` directories. These demos are completely coded. All of the important concepts involve editing the WSDL file and managing the Artix configuration.

Routing concepts

These demos are similar in design and use the HelloWorld application of the earlier demos. A client process sends the `sayHi` or `greetMe` messages using the SOAP over HTTP protocol. However, rather than sending the request directly to the server process, the request is actually sent to a "switch," or routing, process that redirects the request to the server process. In the protocol routing demo, the transport protocol used between the switch and the server process will be changed from SOAP over HTTP to SOAP over MQ or IIOP tunneling. In the operation routing demo, `sayHi` requests are routed to one server process and `greetMe` requests are routed to a different server process.

The Protocol-Based Routing Demo

This demo is located in the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing` directory. This demo contains four subdirectories: `client`, `factory`, `server`, and `switch`. The `client` and `server` directories contain the corresponding applications. The coding within the client and server processes is identical to the earlier demos and need not be discussed.

The subdirectories `factory` and `switch` include new, routing specific, code. Additionally, each directory includes a private, and slightly different, copy of the `HelloWorld.wsdl` file. To clarify what information is needed by the various processes, each WSDL file includes only the information needed by the code in the same directory. This is not actually required; the processes will only use information that they need and will ignore entries relevant to other processes. When you deploy a routing application, you will probably use both approaches; for example, provide the client process with only the information needed to initiate the request, while the switch and server processes extract required information from a more involved WSDL file.

The Client Process

As noted above, the coding within the `client.cxx` file is unchanged from earlier demos. The accompanying `HelloWorld.wsdl` file is identical to the WSDL file used in the SOAP over HTTP demo. This file includes only the standard namespace declarations within the opening `<definitions>` tag, and the `soap:address` declaration within the `<service>…</service>` tags.

```
<soap:address location="http://localhost:8080"/>
```

As far as the client application is concerned, messages are exchanged with a "server" process on port 8080. In actual fact, the client is communicating with the "switch" or "router" process.

The Server Process

Again, the coding within `server.cxx`, `HelloWorldImpl.h` and `HelloWorldImpl.cxx` is identical to the earlier demos. However, in this demo the server process can use the MQ, IIOP tunneling or HTTP protocols. Consequently, in the `HelloWorld.wsdl` file, the opening `<definitions>` tag

includes both the `mq` and `iiop_tunnel` namespace declarations and the `iiop` or `mq` transport related specifications are nested within the `<service>…</service>` tag.

For the IIOP Tunneling protocol:

```
<iiop:address
      location="file://../tunnel.ior"/>
```

For the MQ protocol:

```
<mq:client  QueueManager="MY_DEF_QM"
                 QueueName="HW_REQUEST"
                 AccessMode="send"
                 ReplyQueueManager="MY_DEF_QM"
                 ReplyQueueName="HW_REPLY"
 />

<mq:server  QueueManager="MY_DEF_QM"
                  QueueName="HW_REQUEST"
                  ReplyQueueManager="MY_DEF_QM"
                  ReplyQueueName="HW_REPLY"
                  AccessMode="receive"
/>
```

Note that the value of the `<service>` tag `name` attribute, and the value of the `<port>` tag `name` attribute, have been changed from the earlier demos. What names you give the service and port are unimportant. What is important is that the names are different from the names of the service and port used by the client application. In this routing application, the client sends messages to the service named HelloWorldService, using the port named HelloWorldPort, which are offered by the switch process. The switch subsequently redirects/routes the message to the service and port offered by the server process.

The Switch Process

If you examine the code in the `switch.cxx` file it will look similar to the code in the `server.cxx` file. The only difference is the call to initialize the underlying runtime,

```
IT_Bus::init (argc, argv, "helloWorldSwitch");
```

which includes a third parameter. Operationally this parameter has the same effect as the `ORBname` parameter used during initialization of the ORB within IONA's Orbix; it specifies a configuration scope that contains configuration information for this specific process.

The HelloWorld.wsdl file

The `HelloWorld.wsdl` file used by the switch process has several significant differences from the WSDL files used by the client and server processes. First, the opening `<definitions>` tag includes a declaration for the routing namespace.

```
xmlns:routing="http://schemas.iona.com/routing"
```

Second, there are four service definitions. The first `<service>…</service>` tag defines HelloWorldService and HelloWorldPort, through which the switch process communicates with the client process, and the second, third and fourth `<service>…</service>` tags define the MQHelloWorldService and MQHelloWorldPort, through which the switch process communicates with the server process (using either HTTP, IIOP_Tunneling, or MQ transports).

Finally, within the `<routing:route>…</routing:route>` tag, the redirection/routing specifics for this switch process are specified.

```
<!-- Routes -->
<routing:route name="r1">
    <routing:source
       service="tns:HelloWorldService" port="HelloWorldPort"/>
    <routing:destination
       service="tns:MQHelloWorldService"
                          port="MQHelloWorldPort"/>
</routing:route>
```

Editing the Configuration File—artix.cfg

You must add configuration information specific to the switch process to the configuration file `<installationDirectory>\artix\1.3\etc\domains\artix.cfg`. At the bottom of the file, append the following:

```
helloWorldSwitch
{
  orb_plugins = ["local_log_stream", "iiop_profile", "giop",
                 "iiop", "bus", "soap", "http", "tunnel", "mq",
                 "routing"];

  event_log:filters = ["*=FATAL+ERROR"];

  plugins:routing:wsdl_url="HelloWorld.wsdl";
};
```

This entry creates a new configuration scope named `helloWorldSwitch`. This scope includes two configuration settings specific to the switch process.

The `orb_plugins` list has been extended to include the `"routing"` plugin. The `"routing"` plugin is part of the Artix product.

The `plugins:routing:wsdl_url` configuration value defines where the switch process can obtain the redirection/routing specifications.

```
plugins:routing:wsdl_url="HelloWorld.wsdl";
```

This value references the `HelloWorld.wsdl` file, which includes the two service and route declarations.

When configured in this way, Artix dynamically converts complex data types from their input representation to their output representation. Alternatively, your application can use a type factory plugin, which provides precompiled class definitions. For this simple demo, there will be no differences in performance. With more complex WSDL types, you may obtain improved performance by using the type factory plugin.

Note: The following section describes an optional alternative approach that may provide increased performance. These demos do not require use of this option.

Generating the it_demo_helloWorld_type_factory library

The type factory, "helloworld_typefactory", plugin is created in the factory directory. It is a plugin that supplies the switch process with the capability to create complex, application specific datatypes. In this application, there are no complex datatypes, and this plugin does not provide any functionality.

To use this plugin, you must add it to the orb_plugins listing in the helloWorldSwitch scope.

```
orb_plugins = [“local_log_stream”, “iiop_profile”, “giop”,
                 “iiop”, “bus”, “soap”, “http”, “tunnel”, “mq”,
                 “routing”, "helloworld_typefactory"];
```

You must also add two additional configuration variables to the helloWorldSwitch scope.

The plugins:routing:use_type_factory configuration value must be assigned the value "true" and the plugins:helloworld_typefactory:shlib_name configuration value provides the name of the library file that includes the executable code for the “helloworld_typefactory” plugin.

```
plugins:routing:use_type_factory="true";

plugins:helloworld_typefactory:shlib_name=
   “it_demo_helloworld_type_factory”;
```

This library file is created in the factory subdirectory during the compilation process. All of the information needed to create this library is contained within the HelloWorld.wsdl file included in the factory subdirectory.

Note: The naming convention used for the library entry is platform independent. That is, the value of this configuration variable does not indicate what platform or compiler was used to create the library. The actual name of the library file does include this information. The Artix runtime is aware of platform and compiler restrictions and uses this knowledge and the value of the plugins:…:shlib_name variable to identify the corresponding library file.

Examine the WSDL file in the `factory` subdirectory and note that it contains only one `<service>…</service>` tag and does not include the `<routing:route>…</routing:route>` tag. As far as the type factory library is concerned, the important information is contained within the `<types>…</types>`, `<portType>…</portType>` and `<binding>…</binding>` tags, which are identified through the entries within the `<service>…</service>` tag. Since all of the WSDL files used by this application include the same `<portType>…</portType>` and `<binding>…</binding>` tag content, either `<service>…</service>` tag would have been acceptable. Which `<service>…</service>` tag you choose does, however, affect the compilation process, as you will see in the following section.

Compiling and Running the Application

Before you can fully appreciate the what is happening during the compilation process, you need to review the what happens when you process the WSDL file with the `wsdltocpp.bat` file.

The `wsdltocpp.bat` file may use any, or all, of the following command line arguments.

```
[-e Web-service-name] [-t port] [-b binding-name]
[-d output-directory] [-n namespace] [-f]
[-impl]
[-v] [-license] [-?]
```

Previous demos used only the `-w`, `-n` and `-d` command line arguments. For this demo, you will also use the `-e`, `-t`, and `-f` command line arguments.

The WSDL files currently included in the `client` and `server` subdirectories contain only one `<service>...</service>` tag and, as in the previous demos, the `wsdltocpp.bat` utility can determine what service and port need to be supported in the generated code. However, in this demo the WSDL files were edited so that they only included information relevant to the associated process. You may want to use a common WSDL file for all processes within your application or your routing paradigm require that multiple `<service>...</service>` tags be included in the WSDL files associated with each process. In these situations, you need to specify which service should be referenced from within the generated code. You do this via the `-e` and `-t` command line arguments. To demonstrate usage, the `makefile` in each subdirectory includes these arguments.

What is the purpose of the `-f` command line argument? Including the `-f` argument causes the `wsdltocpp.bat` utility to generate the type factory library file. Consequently, the `makefile` in the `factory` subdirectory includes this argument.

Note: If you have chosen to use the type factory approach to routing, the switch process needs access to this library file. You must place the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing\factory` subdirectory on the `PATH` before running the application.

Compiling the Application Code

Since all of the subdirectories already contain the required files, you can simply compile the application from the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing` directory.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing` directory and issue the command

   ```
   nmake all
   ```

Running the Application

You must first start the server and switch processes and then run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `batch file artix_env[.bat]`.
2. [Only required if you have chosen to use the type factory approach to routing.]

   Add the `factory` subdirectory to the `PATH` by issuing the command:

   ```
   set PATH=%PATH%;<installationDirectory>\artix\1.3\demos
                       \routing\protocol_routing\factory
   ```
3. Move to the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing\server` subdirectory and issue the command:

   ```
   start server
   ```
4. Move to the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing\switch` subdirectory and issue the command:

   ```
   start switch
   ```
5. Move to the `<installationDirectory>\artix\1.3\demos\routing\protocol_routing\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows.

Using other transport protocols

Each of the WSDL files includes a `<service>…</service>` entry that specifies a SOAP over HTTP port definition for the MQHelloWorldService. Alternatively, communication between the switch and server processes can be SOAP over MQ or IIOP Tunneling.

In addition, when starting the server process that receives requests via IIOP tunneling, you must specify appropriate configuration information. As described in the IIOP tunneling example, the configuration information is included in the scope `tunnel.demo`. Consequently, start the server process with the command:

```
start server -ORBname tunnel.demo
```

Stopping the processes

Stop the server and switch processes by giving the `Ctrl-C` command in their respective command windows.

Understanding the Application

This is a basic illustration of how to propagate an invocation across multiple transport protocols. The WSDL file used by the client application includes one `<service><port>` section that specifies what protocol (SOAP/HTTP) and port (8080) the client application must use to communicate with the switch process.

The WSDL file used by the server application includes one `<service><port>` section that specifies what protocol (SOAP/MQ, SOAP/IIOP_Tunneling, or SOAP/HTTP), and related connection information, the server process will use to receive requests from the switch process.

Note: Although the equivalent of multiple `<service><port>` sections exist, only one is active. The alternative connection information is commented out.

The WSDL file used by the switch application includes two active `<service><port>` sections; the first specifies communication between the client and switch processes while the second describes communication between the switch and server processes.

In addition, the WSDL file used by the switch application includes the `<route>` section, which specifies that requests received via SOAP/HTTP on port 8080 should be routed to the server using an alternative protocol.

All requests from the client are sent to the server process via the alternative protocol. In the following demo, you will employ a more complex routing paradigm in which specific operation invocations are routed to different server processes.

The Operation-Based Routing Demo

This demo is located in the `<installationDirectory>\artix\1.3\demo\routing\operation_routing` directory. This demo contains five subdirectories: `client`, `factory`, `server`, `server2`, and `switch`.

This demo is identical to the protocol routing demo with the exception that you will start two server processes. The switch process will redirect/route the `sayHi` messages to the first server process and redirect/route the `greetMe` messages to the second server process.

The only differences between this demo and the protocol routing demo is in the WSDL files associated with the switch and server processes.

The switch process WSDL file

As with the protocol routing demo, multiple `<service>...</service>` entries are used to specify the transports between the client and switch processes and the switch and server processes. In this demo,

- The first `<service>...</service>` tag defines the communication between the client and switch processes – SOAP over HTTP – service name HelloWorldService, with the switch using port 8080.
- The second `<service>...</service>` tag defines the communication between the switch process and the first instance of the server – SOAP over HTTP – service name MQHelloWorldService, with the server using port 8090.
- The third `<service>...</service>` tag defines the communication between the switch process and the second instance of the server – SOAP over HTTP – service name RTHelloWorldService, with the server using port 8085.

Since the switch process needs to redirect/route `sayHi` messages to the first server process, and `greetMe` messages to the second server process, there are two `<routing:route>...</routing:route>` tags. In addition to the `<routing:source>` and `<routing:destination>` tags, the `<routing:operation>` tag specifies which messages should be redirected/routed to the destination service.

These specifications are summarized in the following extract from the WSDL file.

```
<service name="HelloWorldService">
    <port binding="tns:HelloWorldPortBinding"
          name="HelloWorldPort">
      <soap:address location="http://localhost:8080"/>
    </port>
</service>

<service name="RTHelloWorldService">
    <port binding="tns:HelloWorldPortBinding"
          name="RTHelloWorldPort">
      <soap:address location="http://localhost:8085/"/>
    </port>
</service>

<service name="MQHelloWorldService">
    <port binding="tns:HelloWorldPortBinding"
          name="MQHelloWorldPort">
      <soap:address location="http://localhost:8090/"/>
    </port>
</service>

<routing:route name="r1">
    <routing:source service="tns:HelloWorldService"
                    port="HelloWorldPort"/>
    <routing:operation name="sayHi"/>
    <routing:destination service="tns:MQHelloWorldService"
                         port="MQHelloWorldPort"/>
</routing:route>

<routing:route name="r2">
    <routing:source service="tns:HelloWorldService"
                    port="HelloWorldPort"/>
    <routing:operation name="greetMe"/>
    <routing:destination service="tns:RTHelloWorldService"
                         port="RTHelloWorldPort"/>
</routing:route>
```

The Server Process WSDL file

The WSDL files associated with each of the server processes are identical with the exception of the `<service>...</service>` tag, which specifies a unique service name and port. The first server process uses the service

name MQHelloWorldService and listens on port 8090. The second server process uses the service name RTHelloWorldService and listens on port 8085.

The client process WSDL file

The client process has no knowledge of the redirection/routing performed by the switch. Its WSDL file includes only the `<service>...</service>` tag that specifies the HelloWorldService and SOAP over HTTP communication with the switch on port 8080.

Compiling and Running the Application

Since all of the subdirectories already contain the required files, you can simply compile the application from the `<installationDirectory>\artix\1.3\demos\routing\operation_routing` directory.

Compiling the Application Code

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\routing\operation_routing` directory and issue the command

   ```
   nmake all
   ```

Running the Application

You must first start the server process and then run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the batch file `artix_env[.bat]`.
2. [Only required if you have chosen to use the type factory approach to routing.]

   Add the `factory` subdirectory to the `PATH` by issuing the command:

   ```
   set PATH=%PATH%;<installationDirectory>\artix\1.3\demos
                       \routing\operation_routing\factory
   ```
3. Move to the `<installationDirectory>\artix\1.3\routing\operation_routing\server` subdirectory and issue the command:

   ```
   start server
   ```
4. Move to the `<installationDirectory>\artix\1.3\demos\routing\operation_routing\server2` subdirectory and issue the command:

   ```
   start server
   ```
5. Now move to the `<installationDirectory>\artix\1.3\demos\routing\operation_routing\switch` subdirectory and issue the command:

   ```
   start switch
   ```

6. Finally move to the `<installationDirectory>\artix\1.3\demos\routing\operation_routing\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows. Note that each server processes only handles a single message. Stop the server and switch processes.

Stopping the processes

Stop the server and switch processes by giving the `Ctrl-C` command in their respective command windows.

Understanding the Application

This operation routing demo is a little more complex than the protocol routing demo. In this example, the client process uses SOAP/HTTP to communicate with the switch process, which is listening on port 8080. If you examine the WSDL file included in the `client` subdirectory, you will note that there is only one `<service><port>` section, which provides this connection information.

The server processes listen for SOAP/HTTP requests from the switch process on port 8085 or 8090, respectively. If you examine the WSDL file included in the `server` and `server2` directories, you will note that there is only one `<service><port>` section in each file, which provides the relevant connection information for that server.

The WSDL file within the `switch` directory is more complex. This file includes three `<service><port>` sections. The first entry, including SOAP/HTTP and port 8080, specifies what protocol and port the switch process uses to receive requests from the client process; this is the same specification as contained in the WSDL file included in the `client` directory. The second entry, including SOAP/HTTP and port 8085, specifies what protocol and port the switch process uses to send requests to one of the server processes, while the third entry specifies the same information for the other server process. These entries are the same as the corresponding sections in the WSDL files included in the `server` and `server2` directories.

In addition, the WSDL file used by the switch process includes multiple `<route>` sections that specify which operation invocations should be directed to each server process.

Monitoring the Runtime Environment

After starting the first server process, open a command window and issue the command

```
netstat -a
```

Review the listing and confirm that port 8090 is listed. Now start the second server process and again review the port usage; confirm that port 8085 is also listed. Finally, start the switch process and review the port usage; confirm that port 8080 is included in the list.

Embedding the Switch Functionality in a Process

In the protocol routing and operation routing demos, request routing was performed by a switch process that runs independently of the client and server processes. If you examine the source code in the `switch.cxx` file, it is difficult to determine where the routing processing logic is coded. The application simply initializes the runtime environment and enters a processing loop.

```
using namespace IT_Bus;

int
main(int argc, char* argv[]
)
{
    try
    {
        IT_Bus::init(argc, argv, “helloWorldSwitch”);

        IT_Bus::run();
    }
    catch (IT_Bus::Exception& e)
    {
        printf(“Exception occurred: %s”, e.Message());
        return 1;
    }
    return 0;
}
```

In actual fact, the “business end” of this application is performed by the application specific and routing plugins that you added to the `orb_plugins` list that this application uses. Recall that before you ran either demo, you created the `helloWorldSwitch` configuration scope, which includes three runtime specifications.

```
helloWorldSwitch
{
  orb_plugins = [“local_log_stream”, “iiop_profile”, “giop”,
                 “iiop”, “bus”, “soap”, “http”, “mq”, “tunnel”,
                 “routing”];

  event_log:filters = [“*=FATAL+ERROR”];

  plugins:routing:wsdl_url=”HelloWorld.wsdl”;

};
```

The `orb_plugins` list includes the `routing` plugin, which is part of the Artix product.

The `event_log:filters` entry allows the logging level to be set at the application level. The `plugins:routing:wsdl_url` entry provides the path to the WSDL file that includes the routing related entries, i.e., the multiple `<service><port>` sections and the `<route>` section.

By starting the switch process under the `helloWorldSwitch` configuration scope, the plugins that provide routing functionality are automatically loaded.

It would seem, therefore, that since routing is implemented through plugins, simply loading the required plugins during process initialization should provide routing functionality. This is, fortunately, the case, as is illustrated in the next sections.

Modifying the HelloWorld.wsdl File

Recall that each application uses a different version of the `HelloWorld.wsdl` file. This is not actually required.

During code generation, you specified what service and port corresponded to each process. For example, if you examine the `makefile` in the `client` subdirectory you will see that the client process runs against the HelloWorldService/HelloWorldPort endpoint. This is specified through the `-e` and `-t` flags to the `wsdltocpp` utility. The server process in the `server`

subdirectory runs as the MQHelloWorldService/MQHelloWorldPort endpoint while the server process in the `server2` subdirectory runs as the RTHelloWorldService/RTHelloWorldPort endpoint.

Since the endpoint specifications are encoded into the classes generated by the `wsdltocpp` utility, the stub and skeleton classes will only access the `<service><port>` information corresponding to the specified endpoint. The fact that the WSDL file includes multiple `<service><port>` sections for multiple endpoints is immaterial.

Alter the WSDL file

In the operation routing demo, copy the `HelloWorld.wsdl` file from the `switch` subdirectory into the `client`, `server`, and `server2` directories.

Note: Do not recompile this application. You will change how this application runs through the WSDL file and runtime configuration.

Running the Application

By running an application under the `helloWorldSwitch` configuration scope, you enable basic routing functionality. You can specify which process runs the embedded routing functionality by supplying the `-ORBname helloWorldSwitch` command line arguments when starting the process.

Embedding routing functionality within the first server process

Return to the operation routing demo. However, as you start the server processes you will review port usage.

Start the first server process with the command

```
start server -ORBname helloWorldSwitch
```

Open a command window and issue the command

```
netstat -a
```

Review the listing and confirm that both ports 8080 and 8090 are listed.

Start the second server process with the command

```
start server
```

Review the port usage and confirm that port 8085 is also listed.

Run the client process and confirm that requests are properly routed to the two servers.

Stop the server processes.

Embedding routing functionality within the second server process

Rerun the operation routing demo. However, as you start the server processes you will review port usage.

Start the first server process with the command

```
start server
```

Open a command window and issue the command

```
netstat -a
```

Review the listing and confirm that port 8090 is listed.

Start the second server process with the command

```
start server -ORBname helloWorldSwitch
```

Review the port usage and confirm that ports 8080 and 8085 are also listed.

Run the client process and confirm that requests are properly routed to the two servers.

Stop the server processes.

Embedding routing functionality within the client process

Rerun the operation routing demo. However, as you start the server processes you will review port usage.

Start the first server process with the command

```
start server
```

Open a command window and issue the command

```
netstat -a
```

Review the listing and confirm that port 8090 is listed.

Start the second server process with the command

```
start server
```

Review the port usage and confirm that port 8085 is also listed.

Run the client process with the command

```
client <someName> -ORBname helloWorldSwitch
```

Confirm that requests are properly routed to the two servers.

Stop the server processes.

The Content-Based Routing Demo

This demo is located in the `<installationDirectory\artix\1.3\demos\routing\content_routing` directory. This demo contains four subdirectories: `client`, `server`, `server2`, and `switch`.

This demo accomplishes the same processing objectives as the operation routing demo, that is, requests for the `sayHi` operation are sent to one server process and requests for the `greetMe` method are sent to a second server process. The difference is that content based routing requires an independent switch process, which now includes processing logic to redirect/route the requests to the appropriate server processes.

In this demo, the switch process hosts an implementation object that implements the same interface (WSDL contract) as the implementation objects in your server processes. Within the switch process, each request is delivered to the implementation object, which then redirects the request based on your content routing paradigm. As you will see, the switch process implementation object creates proxy objects and becomes a client to the Artix server processes.

The HelloWorld.wsdl File

This file includes the same content as the `HelloWorld.wsdl` file used in the operation routing demo with the exception that the `<route>` tag content has been removed. Request routing is no longer the responsibility of the underlying Artix runtime; it will be implemented in code you write into the switch process.

As in the operation routing demo, the `HelloWorld.wsdl` file includes multiple `<service><port>` sections, which specify the ports on which the switch and server processes listen. Again, there is no harm in including all of these entries in the WSDL file used by a process; directives used by the `wsdltocpp` utility during the code generation step insure that each process is properly coded to use a specific `<service><port>` section. However, for clarity, unnecessary `<service><port>` sections are inactive in the WSDL files used by each process.

The Server Applications

The code within the `server` and `server2` subdirectories is identical to the corresponding directories in the previous routing demos. These server processes run completely independently of the switch and client processes. Regardless of the routing paradigm being used, the code within the target objects remains unchanged.

The Client Application

This code is identical to the corresponding application in the previous routing demos. The client process runs completely independently of the switch and server processes. Regardless of the routing paradigm being used, the code within the client application remains unchanged.

The Switch Application

This application differs significantly from the switch processes of the previous routing demos. For protocol or operation routing, a generic switch process, configured to load the routing and application specific plugins, implemented the routing logic. Application specific information was provided through the application specific plugin and the information within the WSDL file `<route>` tag.

For content based routing, the switch process becomes similar to a server process and contains application specific code that provides the routing logic. As an illustration of an alternative coding approach, all of the code for the switch process has been included in a single source code file—`switch.cxx`.

The `switch.cxx` source code file contains three "components": a class definition for the implementation object— `HelloWorldRouterImpl`; a class definition for the implementation class factory object— `HelloWorldRouterImplFactory`; and the process mainline— `main()`. Examine each "component" and observe that they are conceptually identical to the `HelloWorldImpl` and `HelloWorldImplFactory` classes generated by the `wsdltocpp` utility; the `HelloWorldRouterImpl` class implements the same interface (WSDL contract) as the `HelloWorldImpl` class, and the `HelloWorldRouterImplFactory` class implements the same functionality as the `HelloWorldImplFactory` class.

In writing the `HelloWorldRouterImpl` and `HelloWorldRouterImplFactory` classes you are creating code analogous to the proxy code generated by the `wsdltocpp` utility.

The HelloWorldRouterImpl class

This is the most interesting code. Although this class implements the same interface (WSDL contract) as the `HelloWorldImpl` class, the processing logic is completely different.

This class includes two private variables – `sayHiClient` and `greetMeClient` – that are of type `HelloWorldClient`, the proxy class generated by the `wsdltocpp` utility. The constructor code initializes each of these variables; the `HelloWorldRouterImpl` class uses the proxy class in a similar manner as your client application. However, in all of the previous demos, the client code used the default (no argument) constructor when creating an instance of the proxy. This demo uses the overloaded proxy class constructor that allows you to specify the WSDL file, service, and port targeted by the proxy. The `sayHiClient` variable is initialized to use the target endpoint offered by the process started in the `server` subdirectory, whereas the `greetMeClient` variable is initialized to use the target endpoint offered by the process started in the `server2` subdirectory.

```
class HelloWorldRouterImpl: public HelloWorldServer
{
  public:
    HelloWorldRouterImpl(IT_Bus::Bus_ptr bus,
        IT_Bus::Port *port) : HelloWorldServer (bus, port),
        sayHiClient(
            “HelloWorld.wsdl”,
            QName(“”, “MQHelloWorldService”,
                  “http://xmlbus.com/HelloWorld”),
            “MQHelloWorldPort”
        ),
        greetMeClient(
            “HelloWorld.wsdl”,
            QName(“”, “RTHelloWorldService”,
                  “http://xmlbus.com/HelloWorld”),
            “RTHelloWorldPort”
        )
{
}

...

  private:
    HelloWorldClient sayHiClient;
    HelloWorldClient greetMeClient;
};
```

The `sayHi` and `greetMe` method bodies are where you write the content based routing logic, which, in this example, simply invokes the corresponding method on the target endpoint processes. In a more meaningful application, you would implement a more sophisticated routing paradigm.

```
virtual void greetMe
  (const IT_Bus::String& stringParam0,
  IT_Bus::String& Response)
    IT_THROW_DECL((IT_Bus::Exception))
    {
      cout << "HelloWorldRouterImpl::greetMe
              called with message: "
           << stringParam0 << endl;

      IT_Bus::String greetMeResponse;
      greetMeClient.greetMe
        (stringParam0, greetMeResponse);
      Response = greetMeResponse;
    }

virtual void sayHi
  (IT_Bus::String& Response)
    IT_THROW_DECL((IT_Bus::Exception))
    {
      cout << "HelloWorldRouterImpl::sayHi
              called" << endl;

      IT_Bus::String sayHiResponse;
      sayHiClient.sayHi
        (sayHiResponse);
      Response = sayHiResponse;
    }
```

Because the `sayHiClient` and `greetMeClient` proxies are targeted to different endpoints, the `HelloWorldRouterImpl` object created within the switch process will reinvoke the `sayHi` and `greetMe` methods within different server processes.

The HelloWorldRouterImplFactory class

The constructor and `create_server` methods contain the interesting code. In the constructor, the `HelloWorldRouterImplFactory` class registers itself as the target endpoint for the `HelloWorldService`. In the `create_server` method, the factory creates an instance of the `HelloWorldRouterImpl` class, which provides links to the target endpoints `MQHelloWorldService`

and `RTHelloWorldService`. What happens in these methods is directly analogous to the processing within the `HelloWorldImplFactory` class generated by the `wsdltocpp` utility.

```
class HelloWorldRouterImplFactory :
  public ServerFactoryBase
  {
    public:
      HelloWorldRouterImplFactory():
        m_wsdl_location
          ("HelloWorld.wsdl"),
        m_service_name
          ("", "HelloWorldService",
           "http://xmlbus.com/HelloWorld")
      {
        IT_Bus::Bus::register_server_factory(
           m_service_name,
           this
        );
      }

      virtual ~HelloWorldRouterImplFactory()
      {
        IT_Bus::Bus::deregister_server_factory (m_service_name);
      }

      virtual IT_Bus::ServerStubBase*
      create_server(IT_Bus::Bus_ptr bus, IT_Bus::Port *port)
      {
        return new HelloWorldRouterImpl(bus, port);
      }

      virtual const IT_Bus::String &
      get_wsdl_location()
      {
        return m_wsdl_location;
      }

      virtual void
      destroy_server (IT_Bus::ServerStubBase* server)
      {
        delete server;
      }
    private:
      String m_wsdl_location;
      QName m_service_name;
};
```

The switch mainline

The `main` method is similar to the server mainline code except that you must explicitly create an instance of the `HelloWorldRouterImplFactory` class.

```
int
main(int argc, char* argv[])
{
  try
  {
    HelloWorldRouterImplFactory factory;
    IT_Bus::init(argc, argv);
    cout << “Switch service waiting for requests.”
         << endl;
    IT_Bus::run();
  }
  catch (IT_Bus::Exception& e)
  {
    printf(“Exception occurred: %s”, e.Message());
    return 1;
  }
  return 0;
}
```

Compiling and Running the Application

Since all of the subdirectories already contain the required files, you can simply compile the application from the `<installationDirectory>\artix\6.0\demos\routing\contentRouting` directory.

Compiling the Application Code

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\routing\content_routing` directory and issue the command

   ```
   nmake all
   ```

Running the Application

You must first start the server process and then run the client application.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory\artix\1.3\demos\routing\content_outing\server` subdirectory and issue the command:

   ```
   start server
   ```

3. In the command window from step 1, issue the command

   ```
   netstat -a
   ```

   and confirm that port 8090 is active.
4. Move to the `<installationDirectory>\artix\1.3\demos\routing\content_routing\server2` subdirectory and issue the command:

   ```
   start server
   ```

   In the command window from step 1, issue the command

   ```
   netstat -a
   ```

   and confirm that port 8085 is active.
5. Now move to the `<installationDirectory>\artix\1.3\demos\routing\contentRouting\switch` subdirectory and issue the command:

   ```
   start switch
   ```

   In the command window from step 1, issue the command

```
netstat -a
```

and confirm that port 8080 is active.

6. Finally move to the `<installationDirectory>\artix\1.3\demos\routing\content_routing\client` subdirectory and issue the command:

```
client
```

or the command:

```
client "<some name>"
```

Observe the messages in both the server and client command windows. Note that each server processes only handles a single message but that all messages pass through the implementation object within the switch process. Stop the server and switch processes.

Stopping the processes

Stop the server and switch processes by giving the `Ctrl-C` command in their respective command windows.

Understanding the Application

In this example, the client process uses SOAP/HTTP to communicate with the switch process, which is listening on port 8080. If you examine the WSDL file included in the `client` subdirectory, you will note that there is only one `<service><port>` section, which provides this connection information.

The server processes listen for SOAP/HTTP requests from the switch process on port 8085 or 8090, respectively. If you examine the WSDL file included in the `server` and `server2` directories, you will note that there is only one `<service><port>` section in each file, which provides the relevant connection information for that server.

The WSDL file within the `switch` directory is more complex. This file includes three `<service><port>` sections. The first entry, including SOAP/HTTP and port 8080, specifies what protocol and port the switch process uses to receive requests from the client process; this is the same specification as contained in the WSDL file included in the `client` directory. The second entry, including SOAP/HTTP and port 8085, specifies what protocol and port the switch process uses to send requests to one of the server processes, while the third entry specifies the same information for the other server process. These entries are the same as the corresponding sections in the WSDL files included in the `server` and `server2` directories.

The client sends each request to the switch process. The switch process unmarshals the request and makes the corresponding invocation on its implementation object. Content routing logic is applied to each request and the switch process propagates the request, using SOAP/HTTP, to the appropriate server process. The server receives the request, unmarshals, and makes the corresponding invocation on its implementation object. The response from the server implementation object returns to the client via the switch implementation object.

CHAPTER 11

Accessing an Endpoint via Multiple Protocols

In previous chapters, you developed a collection of demos that showed how Artix™ can use a variety of transport protocols to integrate the same client and server applications. You also learned how Artix can function as a middleware switch and reissue a request received on one protocol using a second protocol. In this chapter, you will develop an example that shows how a single Artix server process can use a single implementation object to process requests received via multiple protocols.

In this chapter

This chapter discusses the following topic:

The Common Target Demos

The starting point code for this demo is located in the `<installationDirectory>\artix\1.3\demos\common_target` directory. This demo is completely coded. The important concepts involve editing the WSDL file and one small change to the implementation object.

Example design

In this demo, two client applications send requests to a single Artix server. One client is a C++ CORBA application that uses the CORBA/IIOP transport protocol to send requests; the second client is a C++ application that uses the SOAP/HTTP transport protocol to send requests. The server process receives both types of requests and invokes on a common implementation object. Neither client is aware of, nor cares about, the other.

The WSDL file now includes two `<binding>` and `<service>` specifications, one for the CORBA/IIOP transport and one for the SOAP/HTTP transport. Both `<binding>` specifications refer to the same `<portType>` specification, so the operations available to each client are similar.

The HelloWorld.wsdl File

There is nothing new in this file. Although there are two `<binding>` and `<service>` specifications, these sections simply contain copies of the equivalent sections of earlier demos. As in the routing demos of the previous chapter, you will need to use the `-e` and `-t` command line arguments to the `wsdltocpp` utility when processing the WSDL file.

The <definition>, <types> and <portType> specifications

These sections of the WSDL document are identical to the corresponding sections within the WSDL document used in the CORBA/IIOP transport demos.

```
<?xml version="1.3" encoding="UTF-8"?>
<definitions name="HelloWorld.idl"
   targetNamespace=
      "http://schemas.iona.com/idl/HelloWorld.idl"
   xmlns=
      "http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap=
      "http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns=
      "http://schemas.iona.com/idl/HelloWorld.idl"
   xmlns:xsd=
      "http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1=
      "http://schemas.iona.com/idltypes/HelloWorld.idl"
   xmlns:corba=
      "http://schemas.iona.com/bindings/corba"
   xmlns:corbatm=
      "http://schemas.iona.com/bindings/corba/typemap">

   <types>
     <schema targetNamespace=
          "http://schemas.iona.com/idltypes/HelloWorld.idl"
       xmlns=
          "http://www.w3.org/2001/XMLSchema"
       xmlns:wsdl=
          "http://schemas.xmlsoap.org/wsdl/">
         <element name="HW.HelloWorld.sayHi.return"
                       type="xsd:string"/>
         <element name="HW.HelloWorld.greetMe.user"
                       type="xsd:string"/>
         <element name="HW.HelloWorld.greetMe.return"
                       type="xsd:string"/>
     </schema>
   </types>
   <message name="HW.HelloWorld.sayHi"/>
   <message name="HW.HelloWorld.sayHiResponse">
         <part name="return"
                  element="xsd1:HW.HelloWorld.sayHi.return"/>
   </message>
```

```
   <message name="HW.HelloWorld.greetMe">
         <part name="user"
                element="xsd1:HW.HelloWorld.greetMe.user"/>
   </message>
   <message name="HW.HelloWorld.greetMeResponse">
         <part name="return"
                element="xsd1:HW.HelloWorld.greetMe.return"/>
   </message>

   <portType name="HW.HelloWorld">
         <operation name="sayHi">
           <input message="tns:HW.HelloWorld.sayHi"
                 name="sayHi"/>
           <output message="tns:HW.HelloWorld.sayHiResponse"
                 name="sayHiResponse"/>
         </operation>
         <operation name="greetMe">
           <input message="tns:HW.HelloWorld.greetMe"
                 name="greetMe"/>
           <output message="tns:HW.HelloWorld.greetMeResponse"
                 name="greetMeResponse"/>
         </operation>
   </portType>
...
</definitions>
```

The <binding> specification

There are two `<binding>` specifications; the first describes the binding within the `corba` namespace and the second describes the binding in the `soap` namespace. The CORBA specification is identical to the corresponding specification in the CORBA/IIOP demo and the soap specification is identical to the corresponding specification in the SOAP/HTTP demo.

```
<binding name="HW.HelloWorldBinding"
    type="tns:HW.HelloWorld">
  <corba:binding repositoryID="IDL:HW/HelloWorld:1.3"/>
  <operation name="sayHi">
    <corba:operation name="sayHi">
      <corba:return name="return"
            idltype="corba:string"/>
    </corba:operation>
    <input/>
    <output/>
  </operation>
```

```
  <operation name="greetMe">
    <corba:operation name="greetMe">
      <corba:param name="user"
            mode="in" idltype="corba:string"/>
      <corba:return name="return"
            idltype="corba:string"/>
    </corba:operation>
    <input/>
    <output/>
  </operation>
</binding>

<binding name="SOAPHelloWorldPortBinding"
    type="tns:HW.HelloWorld">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="greetMe">
    <soap:operation soapAction="" style="rpc"/>
      <input name="greetMe">
        <soap:body encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"
          namespace=
              "http://xmlbus.com/HelloWorld" use="encoded"/>
      </input>
      <output name="greetMeResponse">
        <soap:body encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"
          namespace=
              "http://xmlbus.com/HelloWorld" use="encoded"/>
      </output>
  </operation>
  <operation name="sayHi">
    <soap:operation soapAction="" style="rpc"/>
      <input name="sayHi">
        <soap:body encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"
          namespace=
              "http://xmlbus.com/HelloWorld" use="encoded"/>
      </input>
      <output name="sayHiResponse">
        <soap:body encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"
          namespace=
              "http://xmlbus.com/HelloWorld" use="encoded"/>
      </output>
  </operation>
</binding>
```

The <service> specifications

There are two <service> specifications. The CORBA specification is identical to the corresponding specification in the CORBA/IIOP demo and the soap specification is identical to the corresponding specification in the SOAP/HTTP demo.

```
<service name="SOAPHelloWorldService">
  <port binding="tns:SOAPHelloWorldPortBinding"
        name="SOAPHelloWorldPort">
    <soap:address location="http://localhost:8080"/>
  </port>
</service>

<service name="HW.HelloWorldService">
  <port binding="tns:HW.HelloWorldBinding"
        name="HW.HelloWorldPort">
    <corba:address location=
        "file://../HelloWorld.ior"/>
<!--
    <corba:address location=
        "corbaname:rir:/NameService#helloWorld"/>
-->
  </port>
</service>
```

The Demo Code

There are no changes to the `client.cxx`, `server.cxx`, or `HW_HelloWorldImpl.h` files. Edits have been made to the `HW_HelloWorldImpl.cxx` file.

Modifying the Implementation Class Code

If you run the `wsdltocpp` utility with a WSDL file containing multiple `<service>` specifications (as is true in this demo), you must use the `-e` and `-t` flags to indicate which `<service>` the implementation object should represent. When you use the `-impl` flag, the `wsdltocpp` utility will also generate starting point code for your implementation object; the `<service>` specified through the `-e` and `-t` flags will be incorporated into the generated code. Yet, in this demo, you want the implementation object to represent both `<service>` specifications. Consequently, you must edit some of the generated code within the `HW_HelloWorldImpl.cxx` file. This edit has no effect on the method bodies for your business methods; the editing involves the processing logic within the constructor of the factory for your implementation class.

```
HW_HelloWorldImplFactory::HW_HelloWorldImplFactory()
{
    m_wsdl_location =
      IT_Bus::String("HelloWorld.wsdl");
    IT_Bus::QName service_name
      ("", "HW.HelloWorldService",
       "http://schemas.iona.com/idl/HelloWorld.idl");
    IT_Bus::Bus::register_server_factory(
      service_name,
      this
    );
// Register a second service with the same implementation object
    IT_Bus::QName service_nameSOAP
      ("", "SOAPHelloWorldService",
       "http://schemas.iona.com/idl/HelloWorld.idl");
    IT_Bus::Bus::register_server_factory(
      service_nameSOAP,
      this
    );
}
```

A second service registration is provided through the added code. This edit has already been made to the demo's code.

Compiling the Application Code

Since all of the subdirectories already contain the required files, you can simply compile the application from the `<installationDirectory>\artix\1.3\demos\common_target` directory.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\common_target` directory and issue the command

   ```
   nmake all
   ```

Running the Application

You must first start the server and then run the client applications.

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\common_target\artix_server` subdirectory and issue the command:

   ```
   start server
   ```
3. Move to the `<installationDirectory>\artix\1.3\demos\common_target\corba_client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```
4. Move to the `<installationDirectory>\artix\1.3\demos\common_target\artix_client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

Observe the messages in both the server and client command windows. Invocations from both client applications are going to the same Artix server and implementation object.

Terminating the server process

Issue the `Ctrl-C` command in the corresponding command window.

View the object reference

Note that a string version of the CORBA object reference is written to the subdirectory `<installationDirectory>\artix\1.3\demos\common_target`.

Move to this subdirectory and issue the command

```
iordump helloWorld.ior
```

Confirm that the repository ID represented by this object reference is the same as specified in the WSDL file.

CHAPTER 12

Oneway Operations

This chapter describes how you specify oneway operations.

In this chapter

This chapter discusses the following topics:

Web Service Semantics

Using the `<portType>` section of the WSDL file, you can specify the semantics of your Web service operations. WSDL defines four types of operations:

- Oneway Operation: The client process sends a message to the server process; a corresponding return message from the server process to the client process is not expected. The client process resumes processing immediately after sending the message.
- Request-Response Operation: The client process sends a message to the server process; the server process sends a corresponding return message to the client process. The client process blocks until the return message is received.
- Solicit-Response Operation: The server process sends a message to the client process; the client process sends a corresponding return message to the server process. The server process blocks until the return message is received.
- Notification Operation: The server process sends a message to the client process; a corresponding return message from the client process to the server process is not expected. The server process resumes processing immediately after sending the message.

Up to this point in the tutorial, all operations have been request-response. Consequently, within the `<portType>` section of the WSDL file, each `<operation>` includes both an `<input>` message and an `<output>` message.

Do not be confused by the fact that a `<message>` definition may not include a `<part>` section. For example, a definition for a message that includes a void return, or a message that does not require input or output parameters, would not include `<part>` sections. You have already seen these types of specifications in the `HelloWorld.wsdl` file.

In this chapter you will learn how eliminating the `<output>` tag from an operation's definition specifies oneway semantics.

The WSDL File

To specify oneway versus request-response operations, you need to edit the WSDL file.

The request-response HelloWorld.wsdl file

The `HelloWorld.wsdl` file used in the previous examples included the following `<message>` and `<portType>` specifications.

```
<message name="greetMe">
    <part name="stringParam0"
          type="xsd:string"/>
</message>

<message name="greetMeResponse">
    <part name="return"
          type="xsd:string"/>
</message>

<message name="sayHi"/>

<message name="sayHiResponse">
    <part name="return"
          type="xsd:string"/>
</message>

<portType name="HelloWorldPortType">
    <operation name="greetMe">

        <input message="tns:greetMe"
               name="greetMe"/>
        <output message="tns:greetMeResponse"
               name="greetMeResponse"/>
    </operation>

    <operation name="sayHi">
        <input message="tns:sayHi"
               name="sayHi"/>
        <output message="tns:sayHiResponse"
               name="sayHiResponse"/>
    </operation>

</portType>
```

In this fragment you can see that each operation defined within the `<portType>` section includes both an `<input>` and `<output>` tag (although the input message to the `sayHi` operation does not include a `<part>` representing an input parameter). That is, each operation definition includes a request message (the `<input>` tag) and a response message (the `<output>` tag).

Corresponding to these logical operation definitions, the WSDL file also includes a `<binding>` section that specifies the encoding used for each message. The following fragment is from the WSDL file that describes the SOAP binding.

```
<binding name="HelloWorldPortBinding"
         type="tns:HelloWorldPortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="greetMe">
        <soap:operation soapAction=""
          style="rpc"/>
        <input name="greetMe">
            <soap:body
              encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
              namespace=
                "http://xmlbus.com/HelloWorld" use="encoded"/>
        </input>
        <output name="greetMeResponse">
            <soap:body
              encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
              namespace=
                "http://xmlbus.com/HelloWorld" use="encoded"/>
        </output>
    </operation>
    <operation name="sayHi">
        <soap:operation soapAction=""
          style="rpc"/>
        <input name="sayHi">
            <soap:body
              encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
              namespace=
                "http://xmlbus.com/HelloWorld" use="encoded"/>
        </input>
```

```
        <output name="sayHiResponse">
             <soap:body
               encodingStyle=
                 "http://schemas.xmlsoap.org/soap/encoding/"
               namespace=
                 "http://xmlbus.com/HelloWorld" use="encoded"/>
         </output>
    </operation>
</binding>
```

Note that within the `<binding>`, each `<operation>` includes both an `<input>` and `<output>` designation.

Bindings for other message encodings, for example, TIBCO Rendezvous™ or WebSphere MQ™ contain equivalent entries.

The Generated Code

When this WSDL file is processed by the `wsdltocpp` utility, the following method declarations are created from the operation definitions.

```
virtual void
greetMe(
    const IT_Bus::String & stringParam0,
    IT_Bus::String & var_return
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

virtual void
sayHi(
    IT_Bus::String & var_return
) IT_THROW_DECL((IT_Bus::Exception)) = 0;
```

Note that the `greetMe` method signature contains two parameters: `stringParam0` represents the part associated with the request message `greetMe` and `var_return` represents the part associated with the response message `greetMeResponse`.

The `sayHi` method signature contains only one parameter – `var_return` – that corresponds to the part associated with the response message `sayHiResponse`. The request message `sayHi` did not have a part element and the generated method does not contain a corresponding parameter.

The oneway HelloWorld.wsdl file

To define an operation as oneway, you simply remove references to the output message from the `<message>`, `<portType>` and `<binding>` sections.

```
<message name="greetMe">
    <part name="stringParam0"
          type="xsd:string"/>
</message>

<message name="sayHi"/>

<message name="sayHiResponse">
    <part name="return"
          type="xsd:string"/>
</message>

<portType name="HelloWorldPortType">
    <operation name="greetMe">
        <input message="tns:greetMe"
               name="greetMe"/>
    </operation>

    <operation name="sayHi">
        <input message="tns:sayHi"
               name="sayHi"/>
        <output message="tns:sayHiResponse"
                name="sayHiResponse"/>
    </operation>
</portType>
<binding name="HelloWorldPortBinding"
         type="tns:HelloWorldPortType">
    <soap:binding style="rpc"
                  transport=
                    "http://schemas.xmlsoap.org/soap/http"/>
    <operation name="greetMe">
        <soap:operation soapAction=""
                        style="rpc"/>
        <input name="greetMe">
            <soap:body
              encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
              namespace=
                "http://xmlbus.com/HelloWorld" use="encoded"/>
        </input>
    </operation>
```

```
    <operation name="sayHi">
        <soap:operation soapAction="" style="rpc"/>
        <input name="sayHi">
            <soap:body
              encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
              namespace=
                "http://xmlbus.com/HelloWorld" use="encoded"/>
        </input>
        <output name="sayHiResponse">
            <soap:body
              encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
              namespace=
                "http://xmlbus.com/HelloWorld" use="encoded"/>
        </output>
    </operation>

</binding>
```

The WSDL file does not include a definition of a `greetMeResponse` message, and the `greetMe` operation and binding contain only an input message and encoding specification.

In declaring a oneway operation, you remove the `<output>` tags from the `<portType>` and `<binding>` sections.

The Generated Code

When this WSDL file is processed by the `wsdltocpp` utility, the following method declarations are created from the operation definitions.

```
virtual void
greetMe(
    const IT_Bus::String & stringParam0
) IT_THROW_DECL((IT_Bus::Exception)) = 0;

virtual void
sayHi(
    IT_Bus::String & var_return
) IT_THROW_DECL((IT_Bus::Exception)) = 0;
```

Now the `greetMe` method signature only contains a single parameter that represents the part associated with the request message `greetMe`.

Compiling and Running the Application

All the source code and configuration files are in their appropriate directories.

Compiling the Application Code

The `makefile`s include entries that incorporate the files into your executable.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\oneway` directory and issue the command

   ```
   nmake all
   ```

The compilation process creates the `client.exe` and `server.exe` files in their respective directories.

Running the Application

1. Open a command window to the `<installationDirectory>\artix\1.3\bin` directory and run the `artix_env[.bat]` file.
2. Move to the `<installationDirectory>\artix\1.3\oneway\server` directory and issue the command

   ```
   start server
   ```
3. Move to the `<installationDirectory>\artix\1.3\oneway\client` subdirectory and issue the command:

   ```
   client
   ```

   or the command:

   ```
   client "<some name>"
   ```

What you should observe

Within the server code, a 5 second delay has been added to the `greetMe` method body; the code prints a message before and after this delay. Note that the client code completes and the process exits before the processing within the `greetMe` method completes.

This is not the same outcome that would arise from a request-response operation that includes a response message with no part. Although the method signature in the generated code would not include an out parameter (similar to the signature of your oneway `greetMe` method), the client process would block until the processing within the method body completes.

With a oneway operation, the client code does not block until the server-side processing completes.

Terminating the server process

Issue the `Ctrl-C` command in the corresponding command window.

CHAPTER 13

Type Management

In previous chapters you learned how Artix™ can use multiple middleware transports. All of the coding examples were based on a simple HelloWorld application, and String was the only data type used. This chapter provides guidance on how to work with other basic data types as well as complex types created from entries in the WSDL file.

In this chapter

This chapter discusses the following topics:

A More Complex Application

The code contained in the `<installationDirectory>\artix\1.3\demos\simple_client_server` directory implements a more complex application that illustrates how to work with many basic types. Unfortunately, this demo does not cover all of the types defined in the WSDL file. The code described in this chapter presents a more complete review.

The BaseService.wsdl file

This file, located in the `<installationDirectory>\artix\1.3\demos\simple_client_server\server` directory, includes definitions of complex types and messages and operations that use a variety of basic types as well as the complex types.

Do not try to read or completely understand the contents of this file. What is important is the fact that five complex types are defined – `SOAPStruct`, `ArrayOfSOAPStruct`, `ArrayOffloat`, `ArrayOfint`, and `ArrayOfstring` – and that messages and operations include both the basic types as well as these more complex types.

```
<?xml version="1.3" encoding="UTF-8"?>
<definitions name="BaseService"
    targetNamespace="http://soapinterop.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd">

    <types>
        <schema targetNamespace="http://soapinterop.org/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <complexType name="SOAPStruct">
                <all>
                    <element name="varFloat" type="xsd:float"/>
                    <element name="varInt" type="xsd:int"/>
                   <element name="varString" type="xsd:string"/>
                </all>
            </complexType>
```

```
            <complexType name="ArrayOfSOAPStruct">
                <complexContent>
                    <restriction base="SOAP-ENC:Array">
                        <attribute ref="SOAP-ENC:arrayType"
                            wsdl:arrayType="xsd1:SOAPStruct[]"/>
                    </restriction>
                </complexContent>
            </complexType>
            <complexType name="ArrayOffloat">
                <complexContent>
                    <restriction base="SOAP-ENC:Array">
                        <attribute ref="SOAP-ENC:arrayType"
                                wsdl:arrayType="xsd:float[]"/>
                    </restriction>
                </complexContent>
            </complexType>
            <complexType name="ArrayOfint">
                <complexContent>
                    <restriction base="SOAP-ENC:Array">
                        <attribute ref="SOAP-ENC:arrayType"
                                wsdl:arrayType="xsd:int[]"/>
                    </restriction>
                </complexContent>
            </complexType>
            <complexType name="ArrayOfstring">
                <complexContent>
                    <restriction base="SOAP-ENC:Array">
                        <attribute ref="SOAP-ENC:arrayType"
                                wsdl:arrayType="xsd:string[]"/>
                    </restriction>
                </complexContent>
            </complexType>
        </schema>
    </types>

    <message name="echoBase64">
        <part name="inputBase64" type="xsd:base64Binary"/>
    </message>
    <message name="echoBase64Response">
        <part name="return" type="xsd:base64Binary"/>
    </message>
    <message name="echoStruct">
        <part name="inputStruct" type="xsd1:SOAPStruct"/>
    </message>
    <message name="echoStructResponse">
        <part name="return" type="xsd1:SOAPStruct"/>
    </message>
```

```
    <message name="echoStructArray">
        <part name="inputStructArray"
                 type="xsd1:ArrayOfSOAPStruct"/>
    </message>
    <message name="echoStructArrayResponse">
        <part name="return" type="xsd1:ArrayOfSOAPStruct"/>
    </message>
    <message name="echoBoolean">
        <part name="inputBoolean" type="xsd:boolean"/>
    </message>
    <message name="echoBooleanResponse">
        <part name="return" type="xsd:boolean"/>
    </message>
    <message name="echoFloat">
        <part name="inputFloat" type="xsd:float"/>
    </message>
    <message name="echoFloatResponse">
        <part name="return" type="xsd:float"/>
    </message>
    <message name="echoFloatArray">
        <part name="inputFloatArray" type="xsd1:ArrayOffloat"/>
    </message>
    <message name="echoFloatArrayResponse">
        <part name="return" type="xsd1:ArrayOffloat"/>
    </message>
    <message name="echoInteger">
        <part name="inputInteger" type="xsd:int"/>
    </message>
    <message name="echoIntegerResponse">
        <part name="return" type="xsd:int"/>
    </message>
    <message name="echoIntegerArray">
        <part name="inputIntegerArray" type="xsd1:ArrayOfint"/>
    </message>
    <message name="echoIntegerArrayResponse">
        <part name="return" type="xsd1:ArrayOfint"/>
    </message>
    <message name="echoString">
        <part name="inputString" type="xsd:string"/>
    </message>
    <message name="echoStringResponse">
        <part name="return" type="xsd:string"/>
    </message>
```

```
    <message name="echoStringArray">
       <part name="inputStringArray" type="xsd1:ArrayOfstring"/>
    </message>
    <message name="echoStringArrayResponse">
        <part name="return" type="xsd1:ArrayOfstring"/>
    </message>
    <message name="echoDecimal">
        <part name="inputDecimal" type="xsd:decimal"/>
    </message>
    <message name="echoDecimalResponse">
        <part name="return" type="xsd:decimal"/>
    </message>
    <message name="echoDate">
        <part name="inputDate" type="xsd:dateTime"/>
    </message>
    <message name="echoDateResponse">
        <part name="return" type="xsd:dateTime"/>
    </message>
    <message name="echoVoid"/>
    <message name="echoVoidResponse"/>
    <message name="echoHexBinary">
        <part name="inputHexBinary" type="xsd:hexBinary"/>
    </message>
    <message name="echoHexBinaryResponse">
        <part name="return" type="xsd:hexBinary"/>
    </message>

    <portType name="BasePortType">
        <operation name="echoBase64">
            <input message="tns:echoBase64" name="echoBase64"/>
            <output message="tns:echoBase64Response"
                        name="echoBase64Response"/>
        </operation>
        <operation name="echoStruct">
            <input message="tns:echoStruct" name="echoStruct"/>
            <output message="tns:echoStructResponse"
                        name="echoStructResponse"/>
        </operation>
        <operation name="echoStructArray">
            <input message="tns:echoStructArray"
                      name="echoStructArray"/>
            <output message="tns:echoStructArrayResponse"
                        name="echoStructArrayResponse"/>
        </operation>
```

```
        <operation name="echoBoolean">
            <input message="tns:echoBoolean"
                      name="echoBoolean"/>
            <output message="tns:echoBooleanResponse"
                        name="echoBooleanResponse"/>
        </operation>
        <operation name="echoFloat">
            <input message="tns:echoFloat" name="echoFloat"/>
            <output message="tns:echoFloatResponse"
                        name="echoFloatResponse"/>
        </operation>
        <operation name="echoFloatArray">
            <input message="tns:echoFloatArray"
                      name="echoFloatArray"/>
            <output message="tns:echoFloatArrayResponse"
                        name="echoFloatArrayResponse"/>
        </operation>
        <operation name="echoInteger">
           <input message="tns:echoInteger" name="echoInteger"/>
            <output message="tns:echoIntegerResponse"
                        name="echoIntegerResponse"/>
        </operation>
        <operation name="echoIntegerArray">
            <input message="tns:echoIntegerArray"
                      name="echoIntegerArray"/>
            <output message="tns:echoIntegerArrayResponse"
                name="echoIntegerArrayResponse"/>
        </operation>
        <operation name="echoString">
            <input message="tns:echoString" name="echoString"/>
            <output message="tns:echoStringResponse"
                        name="echoStringResponse"/>
        </operation>
        <operation name="echoStringArray">
            <input message="tns:echoStringArray"
                      name="echoStringArray"/>
            <output message="tns:echoStringArrayResponse"
                        name="echoStringArrayResponse"/>
        </operation>
        <operation name="echoDecimal">
            <input message="tns:echoDecimal"
                      name="echoDecimal"/>
            <output message="tns:echoDecimalResponse"
                        name="echoDecimalResponse"/>
        </operation>
```

```
        <operation name="echoDate">
            <input message="tns:echoDate" name="echoDate"/>
            <output message="tns:echoDateResponse"
                        name="echoDateResponse"/>
        </operation>
        <operation name="echoVoid">
            <input message="tns:echoVoid" name="echoVoid"/>
            <output message="tns:echoVoidResponse"
                        name="echoVoidResponse"/>
        </operation>
        <operation name="echoHexBinary">
            <input message="tns:echoHexBinary"
                      name="echoHexBinary"/>
            <output message="tns:echoHexBinaryResponse"
                        name="echoHexBinaryResponse"/>
        </operation>
    </portType>

    <binding name="BasePortBinding" type="tns:BasePortType">
        <soap:binding style="rpc"

   transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="echoBase64">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoBase64">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoBase64Response">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="echoStruct">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoStruct">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoStructResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
        <operation name="echoStructArray">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoStructArray">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoStructArrayResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="echoBoolean">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoBoolean">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoBooleanResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
        <operation name="echoFloat">
            <soap:operation
                  soapAction="http://soapinterop.org/"
   style="rpc"/>
            <input name="echoFloat">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoFloatResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="echoFloatArray">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoFloatArray">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoFloatArrayResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
        <operation name="echoInteger">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoInteger">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoIntegerResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="echoIntegerArray">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoIntegerArray">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoIntegerArrayResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
        <operation name="echoString">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoString">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoStringResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="echoStringArray">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoStringArray">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoStringArrayResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
        <operation name="echoDecimal">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoDecimal">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoDecimalResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="echoDate">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoDate">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoDateResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
        <operation name="echoVoid">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoVoid">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoVoidResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
```

```
        <operation name="echoHexBinary">
            <soap:operation
                  soapAction=
                       "http://soapinterop.org/" style="rpc"/>
            <input name="echoHexBinary">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </input>
            <output name="echoHexBinaryResponse">
                <soap:body
                  encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/"
                  namespace=
                       "http://soapinterop.org/" use="encoded"/>
            </output>
        </operation>
    </binding>

    <service name="BaseService">
        <port binding="tns:BasePortBinding" name="BasePort">
            <soap:address
                location="http://localhost:12345"/>
        </port>
    </service>

</definitions>
```

Also note that in defining these complex types within the `<types>…<\types>` section, and in describing the encoding within the `<binding>…<\binding>` section, specific reference to `SOAP-ENC` data types and the `soap` namespace prefix are used. This WSDL file uses SOAP/RPC semantics. There is s discussion of document/literal semantics in "Comparing SOAP/RPC and Document/Literal Semantics" on page 165.

When you run the `wsdltocpp` utility, these complex types will be represented by class definitions in the files `BaseTypes.h` and `BaseTypes.cxx`. When you write your own applications, you will use the contents of these files to understand the programming interface for your complex types.

Note: In this demo, all of the files generated by the `wsdltocpp` code generation utility are prefixed with the character string `Base`. This prefix is derived from the value assigned to the `name` attribute within the `<portType>` tag. In the `BaseService.wsdl` file, this value is `BasePortType`, which the code generation utility modifies to `Base`. Names ending with `PortType` will be modified; the code generation utility will not modify other names.

The Server Application Code

If you examine the server application code in the `<installationDirectory>\artix\1.3\demos\simple_client_server\server` directory, you will see that the implementation class simply returns the input parameter as each method's output. Consequently, you only need to study the code within the client application to understand how you work with other basic data types and the code generated from your complex type definitions.

The Client Application Code

This code is contained in the `<installationDirectory>\artix\1.3\demos\complex_types\client` directory. This chapter will not discuss the entire application as the approach to coding many of the business methods is similar; rather, the chapter presents an overview of where to find the required application programming interface information.

The complexClient.cxx File

This is the file that represents your client application. At the beginning of the file the header file `BaseClient.h`, which was generated from the WSDL file by the `wsdltocpp` utility, is included in the application. Also, the namespace `IT_Bus` is declared.

```
#include "BaseClient.h"
…
using namespace IT_Bus;
```

These declarations provide access to the class definitions for the base types.

If you now examine the `BaseClient.h` file, you will note that several generated and Artix product-related header files are included.

```
#include "Base.h"
#include <it_bus\service.h>
#include <it_bus\bus.h>
#include <it_bus\types.h>
#include <it_bus\operation.h>
```

For the purposes of this chapter, the two important header files are `types.h` and `Base.h`.

The types.h file

The file `types.h`, which is located in the `<installationDirectory>\artix\6.0\include\it_bus` directory, leads to definitions for the base types. The `types.h` file includes declarations for many of the base types within the `IT_Bus` namespace, such as, Decimal, String, Float, Boolean.

Some of these types are simply `typedef` declarations, and the actual data type is a standard C++ type. For example, Double, Float, Character, Byte and UByte are implemented as standard C++ types.

Other types are `typedef` declarations based on IONA's platform neutral types. For example, the Artix type `IT_Bus::String` is implemented by the type `IT_String` and the `IT_Bus::Short` is implemented by the type `IT_Short`. The definitions of these types are contained in the header files listed at the start of the `types.h` file, and may be found in the subdirectories under the `<installationDirectory>\artix\1.3\include` directory.

If you have a question about using one of these base types, you can find the necessary application programming interface described in one of these header files.

The Base.h File

The file `Base.h` is generated from the WSDL file by the `wsdltocpp` utility. This file includes the class definitions and method signatures for your application. Additionally, this file `#includes` the `BaseTypes.h` file that contains the class definitions and method signatures for your application's complex types. You use the contents of `Base.h` to understand your application's methods and the file `BaseTypes.h` (and its corresponding implementation file `BaseTypes.cxx`) to learn how to work with the application's complex types. This is discussed in greater detail later in this section.

Use the `Base.h` file to determine the signatures for each of the operations originally defined within the `<portType>` section of the WSDL file. If needed, you will use information from the `types.h` file, other Artix product-related header files, and the `BaseTypes.h` and `BaseTypes.cxx` files to manipulate your application's data types.

The BaseTypes.h file

This file contains the C++ class definitions for the objects that represent the complex types defined in the WSDL file. Since this WSDL file employed SOAP/RPC semantics, some of these generated classes, e.g., the array classes, are derived from a template class – `IT_Bus::SoapEncArrayT` – that adheres to the requirements of the SOAP/RPC semantics. Consequently, you will need to reference this class' definition, which is located in the file `<installationDirectory>\artix\1.3\include\it_bus\soap_enc_array.h`.

The BaseTypes.cxx file

This file contains the C++ implementation for those complex types that are not derived from the pre-existing Artix classes that support the SOAP/RPC semantics. In this demo, only the class corresponding to the complex type SOAPStruct is described in this file.

The processing logic

Now that you understand where to find class definitions for both the base types and the code generated from the complex types of your WSDL file, you can understand the processing logic within the `complexClient.cxx` file. This code shows how to create and manipulate each of the base and complex types.

Compiling and Running the Application

This demo includes code only for the client application, which runs against the server application of the `simple_client_server` demo; you must be certain that the server process exists before you can run the application.

Compiling the client application

All of the required files are fully coded.

1. Open a command window and move to the `<installationDirectory>\artix\1.3\bin` directory. Run the batch file `artix_env[.bat]`.
2. Move to the `<installationDirectory>\artix\1.3\demos\complex_types` directory and issue the command

   ```
   nmake all
   ```

The compilation process creates the `client.exe` and the `BaseTypes.h` and `BaseTypes.cxx` files.

Compiling the server application

This application is fully coded, but it is located under the `simple_client_server` directory.

1. From the command window above, move to the `<installationDirectory>\artix\1.3\demos\simple_client_server\server` directory.
2. Issue the command

   ```
   nmake all
   ```

The compilation process (re)creates the `server.exe` file.

Running the application

You first start the server process and then run the client application.

1. From the `<installationDirectory>\artix\1.3\demos\simple_client_server\server` directory, issue the command

   ```
   start server
   ```

2. Move to the `<installationDirectory>\artix\1.3\demos\complex_types\client` directory and issue the command

   ```
   client
   ```

Terminating the server process

Issue the `Ctrl-C` command in the corresponding window.

Comparing SOAP/RPC and Document/Literal Semantics

The WSDL file in the example developed in the previous section used SOAP/RPC semantics. In this section, you will compare the code generated by the `wsdltocpp` utility from two functionally equivalent WSDL files: one file using SOAP/RPC semantics and the other file using document/literal semantics.

The WSDL files for this example are located in the directory `<installationDirectory>\artix\1.3\demos\complex_types\wsdl`. The file `soaprpc.wsdl`, in the subdirectory `soap_rpc`, represents the SOAP/RPC encoding; the file `docliteral.wsdl`, in the subdirectory `doc_literal`, represents the document/literal encoding.

This example illustrates that the code generated from document/literal semantics is more extensive than SOAP/RPC derived code. Consequently, you may find that WSDL files that use document/literal encoding provide better support for your coding efforts.

Comparing the WSDL files

The first thing you will notice about the two WSDL files is that the `docliteral.wsdl` file is larger than `soaprpc.wsdl` file. If you examine these files in a text editor you will see that there is significantly more content within the `<types>…<\types>` tags, as both base and complex types are defined.

The `<message>…<\message>` and `<portType>…<\portType>` entries are similar, but the contents of the `<binding>…<\binding>` sections are different. This is where the use of SOAP/RPC or document/literal encoding is specified.

Code Generation

You will use the `wsdltocpp` code generation utility and review the contents of the files.

In this example, the `<portType>` is named `InteropPortType`, which the utility shortens to `Interop`. Consequently the files you need to review are `Interop.h`, `InteropTypes.h`, and `InteropTypes.cxx`.

Interop.h file

The two versions of the `Interop.h` file contain the same collection of method signatures. If you look at the signatures for corresponding operations, it appears that the parameter types are different, but this is not really the case. For example, the `echoStruct` signature derived from the SOAP/RPC encoded WSDL file is:

```
virtual void
echoStruct (
  const SOAPStruct $ soapstructParam0,
  SOAPStruct & var_return
) IT_THROW_DECL((IT_Bus::Exception))=0;
```

The corresponding signature derived from the document/literal encoded WSDL file is:

```
virtual void
echoStruct (
  const echoStruct & echoStruct_in,
  echoStructresponse & echoStructResponse_out
) IT_THROW_DECL((IT_Bus::Exception)) = 0;
```

However, if you examine the `<types>…<\types>` section of the `docliteral.wsdl` file, you will observe that the types `echoStruct` and `echoStructresponse` correspond to `SOAPStruct`, so these signatures are actually identical.

You will see the same sort of type substitutions in the method signatures and method bodies in the other files generated by the wsdltocpp utility.

InteropTypes.h file

The file generated from the document/literal encoded WSDL file is significantly larger than the corresponding file from the SOAP/RPC encoded WSDL file. This file now includes declarations for all of the types defined in the `<types>…<\types>` section, both base and complex types.

InteropTypes.cxx file

Again the file generated from the document/literal encoded WSDL file is significantly larger than the corresponding file from the SOAP/RPC encoded WSDL file. This file now includes implementations for all of the methods defined in the `InteropTypes.h` file.