Artix™

Artix for CORBA

Version 3.0, June 2005

Making Software Work Together™

Updated: 30-Oct-2005

Contents

List of Figures

Preface

What is Covered in this Book

This book describes a variety of different CORBA integration scenarios and explains how to use the Artix command-line tools to generate or modify WSDL contracts and IDL interfaces as required. Details of Artix programming, however, do not fall within the scope of this book.

Who Should Read this Book

This book is aimed at engineers already familiar with CORBA technology who need to integrate Web services applications with CORBA.

If you would like to know more about WSDL concepts, see the Introduction to WSDL in *Learning about Artix*.

Finding Your Way Around the Library

The Artix library contains several books that provide assistance for any of the tasks you are trying to perform. The Artix library is listed here, with a short description of each book.

If you are new to Artix

You may be interested in reading:

- Release Notes contains release-specific information about Artix.
- Installation Guide describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- Getting Started with Artix describes basic Artix and WSDL concepts.

To design and develop Artix solutions

Read one or more of the following:

- Designing Artix Solutions provides detailed information about describing services in Artix contracts and using Artix services to solve problems.
- Developing Artix Applications in C++ discusses the technical aspects of programming applications using the C++ API.
- Developing Artix Plug-ins with C++ discusses the technical aspects of implementing plug-ins to the Artix bus using the C++ API.
- Developing Artix Applications in Java discusses the technical aspects of programming applications using the Java API.
- Artix for CORBA provides detailed information on using Artix in a CORBA environment.
- Artix for J2EE provides detailed information on using Artix to integrate with J2EE applications.
- Artix Technical Use Cases provides a number of step-by-step examples of building common Artix solutions.

To configure and manage your Artix solution

Read one or more of the following:

- Deploying and Managing Artix Solutions describes how to deploy Artix-enabled systems, and provides detailed examples for a number of typical use cases.
- Artix Configuration Guide explains how to configure your Artix environment. It also provides reference information on Artix configuration variables.
- IONA Tivoli Integration Guide explains how to integrate Artix with IBM Tivoli.
- IONA BMC Patrol Integration Guide explains how to integrate Artix with BMC Patrol.
- Artix Security Guide provides detailed information about using the security features of Artix.

Reference material

In addition to the technical guides, the Artix library includes the following reference manuals:

- Artix Command Line Reference
- Artix C++ API Reference

- Artix Java API Reference

Have you got the latest version?

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Artix Library

You can search the online documentation by using the **Search** box at the top right of the documentation home page:

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

To search a particular library version, browse to the required index page, and use the **Search** box at the top right. For example:

http://www.iona.com/support/docs/artix/3.0/index.xml

You can also search within a particular book. To search within an HTML version of a book, use the **Search** box at the top left of the page. To search within a PDF version of a book, in Adobe Acrobat, select **Edit**|**Find**, and enter your search text.

Online Help

Artix Designer includes comprehensive online help, providing:

- Detailed step-by-step instructions on how to perform important tasks.
- A description of each screen.
- A comprehensive index, and glossary.
- A full search feature.
- Context-sensitive help.

There are two ways that you can access the online help:

- Click the Help button on the Artix Designer panel, or
- Select **Contents** from the Help menu

Additional Resources

The IONA Knowledge Base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles written by IONA experts about Artix and other products.

The IONA Update Center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to IONA Online Support (http://www.iona.com/support/index.xml).

Comments, corrections, and suggestions on IONA documentation can be sent to docs-support@iona.com.

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`Fixed width`	Fixed 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 `IT_Bus::AnyType` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
`Fixed width italic`	Fixed width 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/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and introduce *new terms*.
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
`%`	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 MS-DOS or Windows 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.
`|`	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in `{}` (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select **File|Open**).

CHAPTER 1

Introduction to CORBA Web Services

Artix provides a flexible framework for bridging between CORBA and Web Services domains. Several different approaches can be used to integrate a CORBA application into a Web Services domain and this introduction provides a brief overview of some typical integration scenarios.

In this chapter

This chapter discusses the following topics:

Artix Architecture

Overview

The key feature of the Artix architecture is that it supports multiple communication protocols. With the help of the plug-in development APIs, moreover, it is possible to extend Artix to support *any* custom protocol.

Figure 1 illustrates this multi-protocol support, showing an Artix application that is capable of sending or receiving operation invocations over three different protocols: SOAP/MQ, SOAP/HTTP, and IIOP.

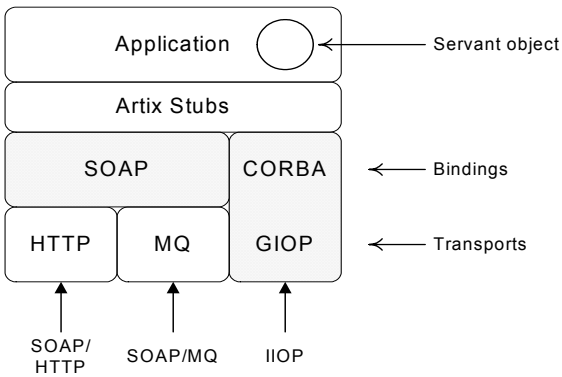

Figure 1: *Artix Application with Multiple Bindings and Transports*

WSDL contract

The Web Services Definition Language (WSDL) contract plays a central role in Artix. It defines the interfaces (or *port types*) and operations for a Web service. In this respect, the WSDL contract is analogous to an IDL interface in CORBA. However, WSDL contracts contain more than just interface definitions. The main elements of a WSDL contract are as follows:

- *Port types*—a port type is analogous to an IDL interface. It defines remotely callable operations that have parameters and return values.
- *Bindings*—a binding describes how to encode all of the operations and data types associated with a particular port type. A binding is specific to a particular protocol—for example, SOAP or CORBA.

- *Port definitions*—a port contains endpoint data that enables clients to locate and connect to a remote server. For example, a CORBA port might contain stringified IOR data.

Servant object

An Artix servant provides the implementation of a port type (analogously to the way in which an Orbix servant provides the implementation of an IDL interface). The servant class is implemented using the appropriate language mapping (an IONA proprietary mapping for C++ or a standard JAX-RPC mapping for Java).

Artix stubs

The Artix stub contains the code that is needed to encode and decode the messages received and sent by an Artix application. Artix provides command-line tools to generate the stub code from WSDL, as follows:

- `wsdltocpp` command—generates C++ stub code from WSDL.
- `wsdltojava` command—generates Java stub code from WSDL.

Bindings

A binding is a particular kind of encoding for operations and data types (for example, CORBA or SOAP). Support for a binding is enabled by loading the relevant plug-in (for example, the `soap` plug-in for SOAP, or the `ws_orb` plug-in for CORBA, and so on).

In addition to loading the relevant plug-in, you must also provide an XML description of the binding in the WSDL contract. Artix provides tools that will generate the binding for you automatically; there is no need to write them by hand.

Transports

A transport is responsible for sending and receiving messages over a specific transport protocol (for example, HTTP or MQ-Series). Support for a transport is enabled by loading the relevant plug-in (for example, the `mq` plug-in for MQ-Series, or the `at_http` plug-in for HTTP).

In Artix, transports are closely associated with port definitions. For example, if you include either a `<http-conf:client/>` or a `<http-conf:server/>` tag within the scope of a `port` element, this indicates that the port uses the HTTP transport.

Artix routers

An Artix router is used to bridge operation invocations between different communication protocols. Figure 2 shows an example of a SOAP/HTTP-to-CORBA router. This router translates incoming SOAP/HTTP request messages into outgoing IIOP request messages. On the reply cycle, the router translates incoming IIOP reply messages into outgoing SOAP/HTTP reply messages.

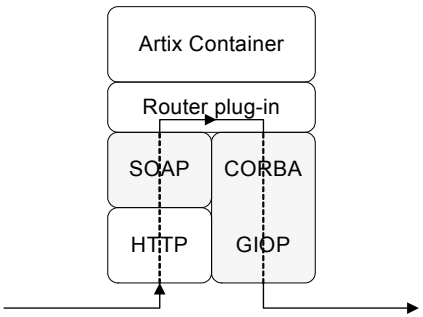

Figure 2: *Example of a SOAP/HTTP-to-CORBA Router*

Artix container

The Artix container, `it_container`, is an application that can be used to run any of the standard Artix services. The functionality of the container is determined by the plug-ins it loads at runtime.

By loading the `router` plug-in (along with the requisite binding and transport plug-ins) the container is configured to run as a standalone router.

Router plug-in

The router plug-in implements a general-purpose protocol bridge. Messages that arrive on one port are sent out on another port.

For example, the router plug-in shown in Figure 2 on page 4 receives request messages over the SOAP/HTTP protocol and forwards the request message out again over the IIOP protocol.

Routes

To configure a router, you need to specify which ports are connected to which other ports. Use the `ns1:route` element to connect a source port to a destination port. For example:

```
<ns1:route name="route_0">
    <ns1:source      service="tns:<SourceService>"
                     port="<SourcePort>"/>
    <ns1:destination service="tns:<DestinationService>"
                     port="<DestinationPort>"/>
</ns1:route>
```

Integrating a CORBA Server with Web Services

Overview

This section considers the problem of a legacy CORBA server that is to be opened up to Web services applications. Artix supports a variety of solutions to this integration problem, which are briefly described in the following subsections.

In this section

This section contains the following subsections:

Accessing the CORBA Server through a Standalone Router

Overview

One of the simplest ways to integrate a WS client with a CORBA server is to deploy a *standalone router* to act as a bridge between them. This approach can be used in any system.

Figure 3 shows a CORBA server that is accessible through a standalone router. The router is responsible for mapping incoming SOAP/HTTP requests into outgoing IIOP requests.

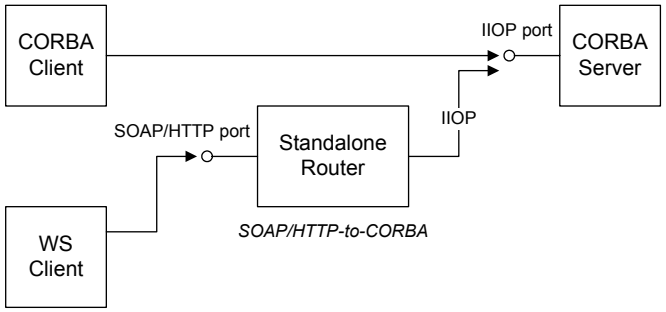

Figure 3: *WS Client Accesses CORBA Server through Standalone Router*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any CORBA server.
- Compatible with any WS client.
- Non-intrusive—no changes need be made either to the client or to the server.

And the following disadvantage:

- Loss of performance—every operation invocation that passes through the router consists of two remote invocations (client-to-router followed by router-to-server).

Accessing the CORBA Server through an Embedded Router

Overview

If the CORBA server is implemented using an Orbix 6.x product, it is usually possible to embed the Artix router directly into the Orbix executable. This approach yields significant performance gains.

Figure 4 shows an example of a CORBA server that is accessible through an embedded router. The router is responsible for mapping incoming SOAP/HTTP requests into colocated IIOP requests.

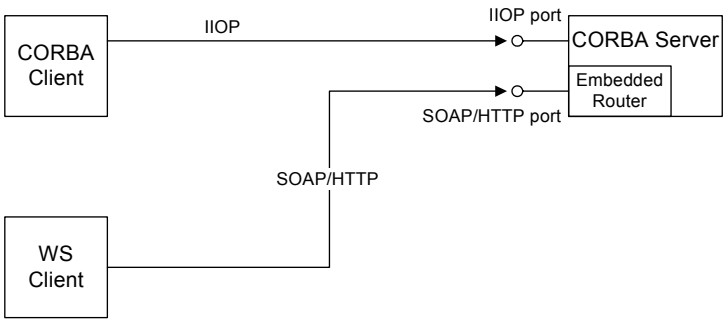

Figure 4: *WS Client Accesses CORBA Server through Embedded Router*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with Orbix 6.x implementations of the CORBA server.
- Compatible with any WS client.
- No changes need be made to the WS client.
- The CORBA server must be reconfigured, but remains otherwise unchanged.

And the following disadvantage:

- Moderate performance—this scenario is more efficient than using a standalone router, but is not as efficient as some other scenarios.

Replacing the WS Client by an Artix Client

Overview

If you have not implemented the WS client yet, you could implement it using Artix. An Artix client offers great flexibility, because it can communicate through multiple protocols, including IIOP and SOAP/HTTP.

Figure 5 shows an example of a CORBA server that is accessed by an Artix client and a CORBA client. The Artix client is configured to talk directly to the CORBA server using the IIOP protocol.

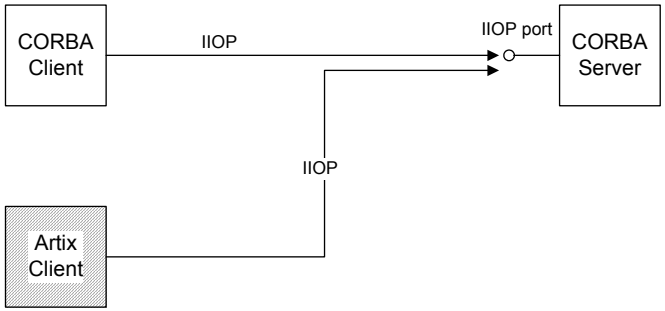

Figure 5: *Replacing the WS Client by an Artix Client*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any CORBA server.
- No changes need be made to the CORBA server.
- Performance is optimized.
- Artix client offers flexibility for future integration.

And the following disadvantage:

- If you have already implemented the WS client, you would have to re-write it to use the Artix APIs.

Replacing the CORBA Server by an Artix Server

Overview

If you want to exploit the full power of the Artix product, you might find it worthwhile to replace the CORBA server by re-implementing it as an Artix server. Because Artix supports multiple protocols, an Artix server can easily support present and future integration requirements.

Figure 6 shows an example of an Artix server that is accessed by a WS client and a CORBA client. The Artix server is configured to accept requests both from CORBA clients and WS clients.

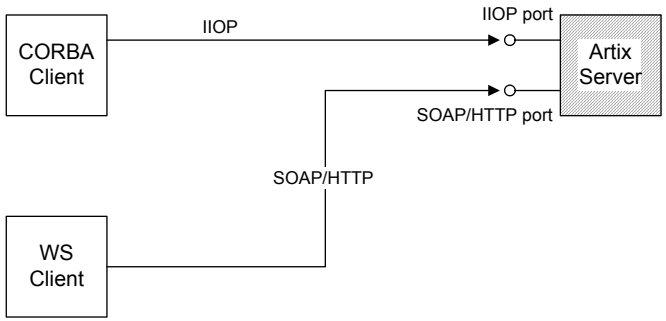

Figure 6: *Replacing the CORBA Server by an Artix Server*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any WS client.
- No changes need be made to the WS client.
- Performance is optimized.
- Artix server offers flexibility for future integration.

And the following disadvantage:

- You must re-implement the CORBA server as an Artix server.

Integrating a CORBA Client with Web Services

Overview

This section considers the problem of CORBA client that needs to access a Web services server. Artix supports a variety of solutions to this integration problem, which are briefly described in the following subsections.

In this section

This section contains the following subsections:

Accessing the WS Server through a Standalone Router

Overview

A relatively simple way to integrate a CORBA client with a WS server is to deploy a *standalone router* to act as a bridge between them. This approach can be used in any system.

Figure 7 shows a WS server that is accessible through a standalone router. The router is responsible for mapping incoming IIOP requests into outgoing SOAP/HTTP requests.

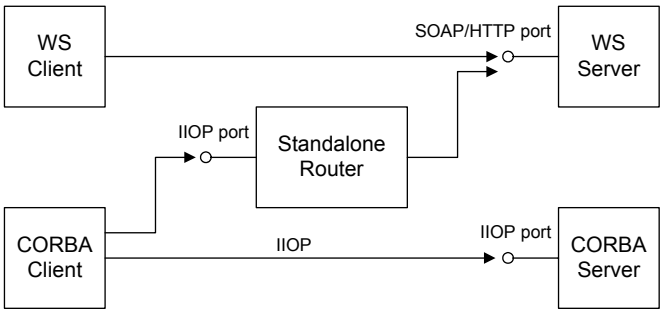

Figure 7: *Client Accesses the WS Server through a Standalone Router*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any WS server.
- Compatible with any CORBA client.
- Non-intrusive—no changes need be made either to the client or to the server.

And the following disadvantage:

- Loss of performance—every operation invocation that passes through the router consists of two remote invocations (client-to-router followed by router-to-server). This has a noticeable impact on performance.

Replacing the CORBA Client by an Artix Client

Overview

To exploit the full power of the Artix product, you might find it worthwhile to replace the CORBA client by re-implementing it as an Artix client. The Artix client can then communicate using a wide variety of protocols, including IIOP and SOAP/HTTP.

Figure 8 shows an example of a WS server that is accessed by an Artix client and a WS client. The Artix client is configured to talk directly to the WS server using the SOAP/HTTP protocol.

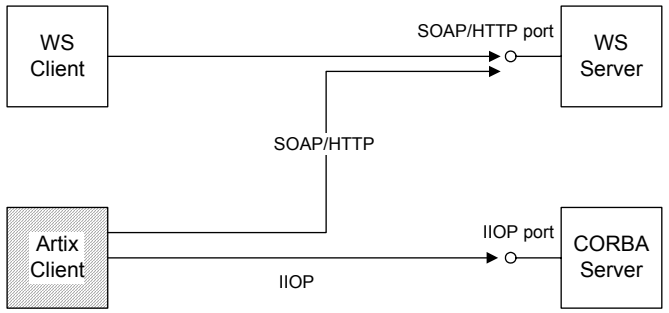

Figure 8: *Replacing the CORBA Client by an Artix Client*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any WS server.
- No changes need be made to the WS server.
- Performance is optimized.
- Artix client offers flexibility for future integration.

And the following disadvantage:

- You must re-implement the CORBA client as an Artix client.

Replacing the WS Server by an Artix Server

Overview

If you want to exploit the full power of the Artix product, you might find it worthwhile to replace the WS server by re-implementing it as an Artix server. Because Artix supports multiple protocols, an Artix server can easily support present and future integration requirements.

Figure 9 shows an example of an Artix server that is accessed by a WS client and a CORBA client. The Artix server is configured to accept requests both from CORBA clients and WS clients.

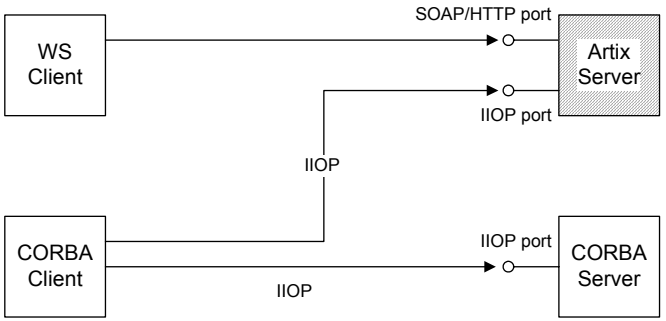

Figure 9: *Replacing the WS Server by an Artix Server*

Advantages and disadvantages

This scenario offers the following advantages:

- Compatible with any CORBA client.
- No changes need be made to the CORBA client.
- Performance is optimized.
- Artix server offers flexibility for future integration.

And the following disadvantage:

- If you have already implemented the WS server using a third-party product, you would have to re-write it as an Artix server.

CHAPTER 2

Exposing a Web Service as a CORBA Service

This chapter describes how to expose a Web service as a CORBA service using Artix. If the Web Service is implemented using Artix, it is relatively easy to integrate with CORBA; if implemented using a third-party product, integration is made possible using Artix routers.

In this chapter

This chapter discusses the following topics:

Converting WSDL to IDL

Overview

To convert a WSDL contract to an equivalent OMG IDL interface (or interfaces), perform the following steps:

1. Add CORBA bindings to WSDL.
2. Add CORBA endpoints to WSDL.
3. Generate the IDL.

Add CORBA bindings to WSDL

Generate a CORBA binding for each port type that you want to expose as an IDL interface:

- If you want to expose a *single* WSDL port type from the WSDL file, `<WSDLFile>.wsdl`, enter the following command:

  ```
  > wsdltocorba -corba -i <PortTypeName> <WSDLFile>.wsdl
  ```

  Where `<PortTypeName>` refers to the `name` attribute of an existing `portType` element. This command generates a new WSDL file, `<WSDLFile>-corba.wsdl`.

- If you want to expose *multiple* WSDL port types, you must run the `wsdltocorba` command iteratively, once for each port type. For example:

  ```
  > wsdltocorba -corba -i <PortType_A> -o <WSDLFile>01.wsdl
     <WSDLFile>.wsdl
  > wsdltocorba -corba -i <PortType_B> -o <WSDLFile>02.wsdl
     <WSDLFile>01.wsdl
  > wsdltocorba -corba -i <PortType_C> -o <WSDLFile>03.wsdl
     <WSDLFile>02.wsdl
  ...
  ```

  Where the `-o` flag is used to specify the name of the output file at each stage. Rename the last file in the sequence to `<WSDLFile>-corba.wsdl`.

Add CORBA endpoints to WSDL

It is not strictly necessary to add CORBA endpoints to the WSDL at this stage (that is, prior to generating the IDL), but it is convenient to make these modifications to the WSDL contract now.

To add the CORBA endpoints, open the *<WSDLFile>*-corba.wsdl file generated in the previous step and add a `service` element for each of the port types you want to expose. For example, a simple CORBA endpoint that is associated with the *<CORBABinding>* binding could have the following form:

```
<definitions name="" targetNamespace="..."
    ...
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    ...>
    ...
    <service name="<CORBAServiceName>">
        <port binding="tns:<CORBABinding>" name="<CORBAPortName>">
            <corba:address location="file:///greeter.ior"/>
        </port>
    </service>
</definitions>
```

The value of the `location` attribute in the `corba:address` element can be specified as one of the following URL types:

- *File URL*—to configure the Artix server to write an IOR to a file as it starts up, specify the `location` attribute as follows:

  `location="file:///<DirPath>/<IORFile>.ior"`

  On Windows platforms, the URL format can indicate a particular drive—for example the `C:` drive—as follows:

  `location="file:///C:/<DirPath>/<IORFile>.ior"`

  Note: It is usually simplest to specify the file name using an absolute path. If you specify the file name using a relative path, the location is taken to be relative to the directory the Artix process is started in, *not* relative to the containing WSDL file.

- *corbaname URL*—to configure the Artix server to bind an object reference in the CORBA naming service, specify the `location` attribute as follows:

  `location="corbaname:rir:/NameService#StringName"`

Where *StringName* is a name in the CORBA naming service. For more details, see "How an Artix Client Resolves a Name" on page 58.

- *Placeholder IOR*—is appropriate for IORs created dynamically at runtime (for example, IORs created by factory objects). In this case, you should use the special placeholder value, `IOR:`, for the location attribute, as follows:

  ```
  location="IOR:"
  ```

  Artix then uses the enclosing `service` element as a template for transient object references.

Note: It is also possible to add a CORBA endpoint to the WSDL contract using the `wsdltoservice` command line tool. For details of this command, see the *Command Line Reference* document.

Generate the IDL

Generate an IDL interface for each port type, as follows:

- To generate IDL for a *single* port type, select the relevant CORBA binding, *<CORBABinding>*, from the WSDL and enter the following command:

  ```
  > wsdltocorba -idl -b <CORBABinding> <WSDLFile>-corba.wsdl
  ```

  The output from this command is written to an IDL file, *<WSDLFile>*`-corba.idl`. If you want to change the name of the IDL output file, you can use the `-o` *<IDLFileName>* option.
- To generate IDL for *multiple* port types, you must run the `wsdltocorba` command once for each port type. After generating all of the IDL interfaces individually, you would typically concatenate the output files into a single IDL file.

Exposing an Artix Web Service as a CORBA Service

Overview

It is relatively straightforward to expose an Artix Web service as a CORBA service. Essentially, you must add the configuration of the relevant CORBA bindings to the WSDL contract and ensure that the requisite CORBA plug-ins are loaded into the Artix application.

In detail, the steps for exposing an Artix service as a CORBA service are as follows:

1. Convert WSDL to IDL.
2. Write code to activate the CORBA endpoints.
3. Re-build the Artix server.
4. Configure the Artix server.

Convert WSDL to IDL

Follow the instructions in "Converting WSDL to IDL" on page 16 to convert your WSDL contract to IDL. The output from this step consists of two files, as follows:

- *Modified WSDL file*—the WSDL contract is modified to include CORBA bindings and CORBA endpoints. The Artix server needs the modified contract to expose the service over CORBA.
- *IDL file*—an IDL file is generated from the modified WSDL. CORBA clients use this IDL file to access the CORBA service exposed by the Artix server.

Write code to activate the CORBA endpoints

In the main function of your application source code, add some code to activate the CORBA endpoints. For example, given the following `service` element in the WSDL contract:

```
<definitions name="" targetNamespace="<TargetNameSpace>"
    ...
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    ...>
    ...
    <service name="<CORBAServiceName>">
        <port binding="tns:<CORBABinding>" name="<CORBAPortName>">
            <corba:address location="..."/>
        </port>
    </service>
</definitions>
```

You can activate all of the ports in the *<CORBAServiceName>* service by registering a servant, as follows:

```
// C++
IT_Bus::QName m_service_qname("", "<CORBAServiceName>",
   "<TargetNameSpace>")

m_bus()->register_servant(
    m_servant,              // Service implementation
    "<WSDLFile>.wsdl",   // WSDL file location
    m_service_qname         // Service QName
);
```

Where `m_servant` is an object that implements a WSDL port type. This could be the very same object that is registered with other protocols, such as SOAP/HTTP, or it could be a new instance of the service. The second argument, *<WSDLFile>*`.wsdl`, gives the location of the modified WSDL contract. In this example, it is assumed that the WSDL contract is stored in the same directory as the application executable.

Note: For more details about activating service endpoints and registering servants, see the “Artix Programming Considerations” chapter from *Developing Artix Applications in C++*.

Re-build the Artix server

Before re-building the Artix server executable, you must regenerate the Artix stub files from the modified WSDL contract. In particular, you must ensure that C++ code is generated for each of the newly-defined CORBA bindings.

After regenerating the stub files, you can re-build the Artix server.

Configure the Artix server

The Artix server must be configured to load the requisite CORBA plug-ins. Example 1 shows how to modify the Artix configuration scope, `artix_srvr_with_corba_binding`, to enable the CORBA bindings.

Example 1: *Artix Configuration Required for a CORBA Binding*

```
# Artix Configuration File

artix_srvr_with_corba_binding {
    ...

    # Modified configuration required for a CORBA binding:
    #
1   orb_plugins = [..., "iiop_profile", "giop", "iiop",
    "ws_orb"];
2   binding:client_binding_list =
    ["OTS+POA_Coloc","POA_Coloc","OTS+GIOP+IIOP","GIOP+IIOP"];

3   plugins:iiop_profile:shlib_name = "it_iiop_profile";
    plugins:giop:shlib_name = "it_giop";
    plugins:iiop:shlib_name = "it_iiop";
    plugins:ws_orb:shlib_name = "it_ws_orb";
};
```

The preceding Artix configuration can be explained as follows:

1. Edit the ORB plug-ins list, adding the plug-ins needed to support CORBA bindings. The following additional plug-ins are needed:
   - `iiop_profile`, `giop`, and `iiop` plug-ins—provide support for the Internet Inter-ORB Protocol (IIOP), which is used by CORBA.
   - `ws_orb` plug-in—enables the Artix application to send and receive CORBA messages.
2. You should ensure that the `binding:client_binding_list` (either within this scope or in the nearest enclosing scope) includes bindings with the `GIOP+IIOP` protocol combination. The client binding list shown here is a typical default setting.

3. For each of the additional plug-ins you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:<plugin_name>:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

   You can optionally specify additional configuration settings for the plug-ins at this point (see the *Artix Configuration Reference* for more details).

Exposing a Non-Artix Web Service as a CORBA Service

Overview

If you want to expose a non-Artix Web service as a CORBA service, you must deploy a standalone Artix router that acts as a bridge between CORBA clients and the Web services server.

In this section

This section contains the following subsections:

Standalone CORBA-to-SOAP Router Scenario

Overview

Figure 10 shows an overview of a standalone CORBA-to-SOAP router. In this scenario, the router is packaged as a standalone application, which acts as a bridge between the CORBA client and the Web services server. The standalone router is responsible for converting incoming CORBA requests into outgoing requests on the Web services server. Replies from the Web services server are converted into CORBA replies by the router and sent back to the client.

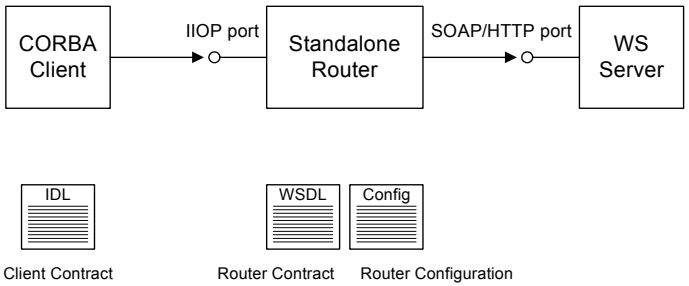

Figure 1: *Standalone Artix Router*

Container

The Artix container, `it_container`, is an executable that can be used to run any of the standard Artix services. The functionality of the container is determined by the plug-ins it loads at runtime.

In this scenario, the container is configured to load the `router` plug-in (along with some other plug-ins) so that it functions as a standalone router.

Modifications to CORBA server

When using a standalone Artix router, no modifications need be made to the CORBA server.

Elements required for this scenario

The following elements are required to implement this scenario:

- IDL interface for clients.
- WSDL contract for the standalone router.
- Artix configuration file for the standalone router.

Configuring and Running a Standalone CORBA-to-SOAP Router

Overview

This section describes how to configure and run a standalone router that acts as a bridge between CORBA clients and a SOAP/HTTP Web services server. The following steps are described:

1. Convert WSDL to IDL.
2. Generate the router.wsdl file.
3. Create the Artix configuration.
4. Run the standalone router.

Convert WSDL to IDL

Follow the instructions in "Converting WSDL to IDL" on page 16 to convert your WSDL contract to IDL and to generate CORBA bindings and CORBA endpoints in the WSDL contract. The output from this step is a modified WSDL file, `<WSDLFile>.wsdl`, and an IDL file.

Generate the router.wsdl file

To generate the `router.wsdl` file, you need to augment the `<WSDLFile>.wsdl` file from the previous step. Specifically, you must add the requisite bindings and endpoints for the second leg of the route, which goes from the router to the SOAP Web service.

1. *Generate CORBA bindings and CORBA endpoints*—if you followed the steps in "Converting WSDL to IDL" on page 16, the `<WSDLFile>.wsdl` file already contains the relevant CORBA bindings and CORBA endpoints.
2. *Generate SOAP bindings*—generate a SOAP binding for each port type that is exposed as an IDL interface. The router acts like a SOAP client with respect to the SOAP Web services server.

   If the router needs to access a *single* WSDL port type, generate a SOAP binding with the following command:

   ```
   > wsdltosoap -i <PortTypeName> -b <BindingName>
     <WSDLFile>.wsdl
   ```

   Where `<PortTypeName>` refers to the `name` attribute of an existing `portType` element and `<BindingName>` is the name to be given to the newly generated SOAP binding. This command generates a new WSDL file, `<WSDLFile>-soap.wsdl`.

If the router needs to access *multiple* WSDL port types, you must run the `wsdltosoap` command iteratively, once for each port type. For example:

```
> wsdltosoap -i <PortType_A> -b <Binding_A>
   -o <WSDLFile>01.wsdl <WSDLFile>.wsdl
> wsdltosoap -i <PortType_B> -b <Binding_B>
   -o <WSDLFile>02.wsdl <WSDLFile>01.wsdl
> wsdltosoap -i <PortType_C> -b <Binding_C>
   -o <WSDLFile>03.wsdl <WSDLFile>02.wsdl
...
```

Where the `-o <FileName>` flag specifies the name of the output file. At the end of this step, rename the WSDL file to `router.wsdl`.

3. Add SOAP endpoints—add a `service` element for each of the port types you want to expose. For example, a simple SOAP endpoint could have the following form:

```
<definitions name="" targetNamespace="..."
    ...
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    ...>
    ...
    <service name="<SOAPServiceName>">
        <port binding="tns:<SOAPBinding>" name="<SOAPPortName>">
            <soap:address location="http://localhost:9000"/>
            <http-conf:client/>
            <http-conf:server/>
        </port>
    </service>
</definitions>
```

In the preceding example, you must add a line that defines the `http-conf` namespace prefix in the `<definitions>` tag.

The most important setting in the SOAP port is the `location` attribute of the `soap:address` element, which can be set to one of the following HTTP URLs:

- Explicit HTTP URL—if a particular service is provided at a fixed address, you can specify the *`<hostname>`* and *`<port>`* values explicitly.

  `location="http://`*`<hostname>`*`:`*`<port>`*

- Placeholder HTTP URL—if a service is created dynamically at runtime, you should specify a transient HTTP URL, as follows:

  `location="http://localhost:0`

  At runtime, the placeholder URL is replaced by an explicit address. Artix then treats the enclosing `service` element as a template, allowing multiple transient services to be created at runtime.

Note: It is also possible to add a SOAP endpoint to the WSDL contract using the `wsdltoservice` command line tool. For details of this command, see the *Command Line Reference* document.

4. Add a route for each exposed port type—for each port type, you need to set up a route to translate incoming CORBA requests into outgoing SOAP requests. For example, the following route definition instructs the router to map incoming CORBA request messages to a SOAP/HTTP endpoint.

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    ...>
    ...
    <ns1:route name="route_0">
        <ns1:source      service="tns:<CORBAServiceName>"
                         port="<CORBAPortName>"/>
        <ns1:destination service="tns:<SOAPServiceName>"
                         port="<SOAPPortName>"/>
    </ns1:route>
</definitions>
```

In the preceding example, you must add a line that defines the `ns1` namespace prefix in the `<definitions>` tag.

The `ns1:source` element identifies the CORBA endpoint in the router that receives incoming requests from a client. The `ns1:destination`

element identifies the SOAP/HTTP endpoint in the Orbix server to which outgoing requests are routed.

> **Note:** Generally, when defining routes, if the `location` of the source endpoint is a placeholder, the `location` of the destination endpoint should *also* be a placeholder.

5. Check that you have added all the namespaces that you need—for a typical CORBA to SOAP/HTTP route, you typically need to add the following namespaces (in addition to the namespaces already generated by default):

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:references="http://schemas.iona.com/references"
    ...>
    ...
</definitions>
```

Create the Artix configuration

Example 2 shows a suitable configuration for a standalone router that maps incoming CORBA requests to outgoing SOAP/HTTP requests.

Example 2: *Artix Configuration Suitable for a Standalone Artix Router*

```
# Artix Configuration File

# Global configuration scope
...

standalone_router {
    # Configuration for standalone router:
    #
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop", "ws_orb", "soap", "at_http", "routing"];

    plugins:routing:wsdl_url="../../etc/router.wsdl";

    plugins:ws_orb:shlib_name = "it_ws_orb";
    plugins:soap:shlib_name = "it_soap";
    plugins:http:shlib_name = "it_http";
```

Example 2: *Artix Configuration Suitable for a Standalone Artix Router*

```
    plugins:at_http:shlib_name = "it_at_http";
    plugins:routing:shlib_name = "it_routing";

    # Uncomment these lines for interoperability with Orbix 3.3
5   #policies:giop:interop_policy:negotiate_transmission_codeset
    = "false";
    #policies:giop:interop_policy:send_principal = "true";
    #policies:giop:interop_policy:send_locate_request = "false";
};
```

The preceding Artix configuration can be explained as follows:

1. The basic configuration settings needed by the Artix container process are inherited from the global configuration scope.
2. Edit the ORB plug-ins, adding the requisite Artix plug-ins to the list. In this example, the following plug-ins are needed:
   - `xmlfile_log_stream` plug-in—enables logging to an XML file.
   - `iiop_profile`, `giop`, and `iiop` plug-ins—enables the IIOP protocol (used by CORBA).
   - `ws_orb` plug-in—enables the router to send and receive CORBA messages.
   - `soap` plug-in—enables the router to send and receive SOAP messages.
   - `at_http` plug-in—enables the router to send and receive messages over the HTTP transport.
   - `routing` plug-in—contains the core of the Artix router.

   If you plan to use other bindings and transports, you might need to add some other Artix plug-ins instead.
3. The `plugins:routing:wsdl_url` setting specifies the location of the router WSDL contract (see "Converting WSDL to IDL" on page 16). The URL can be a relative filename (as here) or a general `file:` URL.
4. To load the Artix plug-ins, you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:`*`<plugin_name>`*`:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

You can also specify additional plug-in configuration settings at this point (see the *Artix Configuration Reference* for more details).

5. If the router needs to integrate with Orbix 3.3 CORBA clients, you should uncomment these lines to enable interoperability. For more details about these configuration settings, see the *Artix Configuration Reference*.

> **Note:** These interoperability settings might also be useful for integrating with other third-party ORB products. See the *Artix Configuration Reference* for more details.

Run the standalone router

Run the standalone router by invoking the container, `it_container`, passing the router's ORB name as a command-line parameter (the ORB name is identical to the name of the router's configuration scope).

For example, to run the router configured in Example 2 on page 28, enter the following at a command prompt:

```
it_container -ORBname standalone_router
```

Using an Orbix 3.3 Client to Access an Artix Server

Overview

This section gives a summary of the problems that might occur when you try to compile an Artix-generated IDL file (generated by the `wsdltocorba` tool) using the Orbix 3.3 IDL compiler.

Because the Orbix 3.3 product was designed to conform to the CORBA 2.1 specification (which is an earlier version of the CORBA specification than that used for Artix) there are some differences between the conventions used in Orbix 3.3 IDL files and the conventions used in Artix IDL files.

Note: The following list of issues is not necessarily exhaustive. This section summarizes only those interoperability issues known about at the time of writing.

Data type compatibility

Most of the IDL data types generated by the Artix `wsdltocorba` tool are compatible with Orbix 3.3. But there are some exceptions. The following WSDL data types require workarounds in order to interoperate with the Orbix 3.3 product:

- xsd:dateType type mapping to the TimeBase::UtcT IDL type.
- Complex type derived from a simple type.

xsd:dateType type mapping to the TimeBase::UtcT IDL type

Artix uses the `TimeBase::UtcT` type to represent the `xsd:dateTime` XML schema type. To support the `TimeBase::UtcT` type, Artix-generated IDL files contain the following `#include` statement:

```
#include <omg/TimeBase.idl>
```

A problem arises, however, when the Orbix 3.3 IDL compiler attempts to compile the `TimeBase.idl` file, because the `TimeBase.idl` file includes `#pragma` macros that are incompatible with the Orbix 3.3 IDL compiler. To fix this problem, perform the following steps:

1. Make a copy of the `TimeBase.idl` file (the original of this file can be found in the *`ArtixInstallDir`*`/artix/`*`Version`*`/idl/omg` directory).
2. Edit the copied file to delete the following #pragma macros:

```
#pragma IT_SystemSpecification

#pragma IT_BeginCBESpecific AllJava                    "@@\
@module TimeBase=org.omg"
```

3. Edit the `#include` statement in the main IDL file, to point at the modified copy of the `TimeBase.idl` file.

Complex type derived from a simple type

A problem arises with XML schema complex types that are defined by derivation from a simple type. For example, consider the following schema type, `Document`, that adds a string attribute to a simple string type:

```
<xsd:complexType name="Document">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="ID" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

When the `wsdltocorba` utility maps this schema type to IDL, it generates the following struct:

```
// IDL
struct Document {
    string_nil ID;
    string _simpleTypeValue;
};
```

When this IDL sample is passed to the Orbix 3.3 compiler, it fails to compile because the Orbix 3.3 compiler does not allow identifiers that begin with the _ (underscore) character.

To work around this problem, you can manually edit the CORBA binding in the WSDL file, replacing `_simpleTypeValue` by `simpleTypeValue` (removing the underscore character). For example, for the `Document` data type, the CORBA binding defines the following mapping by default:

```
<corba:struct name="Document" repositoryID="IDL:Document:1.0"
              type="s:Document">
    <corba:member idltype="ns1:string_nil" name="ID"/>
    <corba:member idltype="corba:string"
                  name="_simpleTypeValue"/>
</corba:struct>
```

To modify the mapping in this case, simply replace `_simpleTypeValue` by `simpleTypeValue` in the preceding code fragment.

Incompatible #pragma macros

The following `#pragma` macros which appear in some standard Artix IDL files are incompatible with Orbix 3.3 and will cause the Orbix 3.3 IDL compiler to report an error:

```
#pragma IT_SystemSpecification
#pragma IT_BeginCBESpecific
```

CHAPTER 3

Exposing a CORBA Service as a Web Service

This chapter describes how to expose a CORBA service as a Web service using Artix. Different approaches can be taken, depending on whether the back-end CORBA service is implemented using the Orbix 6 product, the Orbix 3.3 product or some other third-party ORB product.

In this chapter

This chapter discusses the following topics:

Converting IDL to WSDL

Overview

The first step in exposing a CORBA server as a Web service is to convert the CORBA server's IDL into a WSDL contract. For all of the examples presented in this chapter, the following assumptions are made:

- The server's IDL does not feature callbacks.
- Web service clients use the SOAP/HTTP protocol.

WSDL contract files

This subsection describes how to generate the following two WSDL files:

- `router.wsdl` file—deployed along with the embedded router and the Orbix server, the `router.wsdl` file contains all of the router information required to map incoming SOAP requests to outgoing CORBA requests.
- `client.wsdl` file—contains all of the information required by Web services clients to make SOAP/HTTP invocations on the router.

Contents of the router contract

Given that the router has to be capable of routing incoming SOAP requests to outgoing CORBA requests, the router generally must contain the following elements:

- Port types.
- CORBA bindings.
- SOAP bindings.
- CORBA endpoints.
- SOAP/HTTP endpoints.
- Routes from SOAP/HTTP endpoints to CORBA endpoints.

Generate the router contract

To generate a router contract from a given IDL file, *<IDLFile>*`.idl`, perform the following steps:

1. Generate WSDL from the IDL file—at a command-line prompt, enter:

   ```
   > idltowsdl <IDLFile>.idl
   ```

   This command generates a WSDL file, *<IDLFile>*`.wsdl`, which contains the following:

   - XSD schema types, generated from the IDL data types.

- `portType` elements—a port type for each IDL interface in the source.
- `binding` elements—a CORBA binding for each port type.
- `service` elements—a CORBA endpoint for each port type

You might need to specify additional flags to the `idltowsdl` command utility. Some of the more commonly required options are:

`-r <ref_schema>` specifies the location of the references schema. The schema file, `references.xsd`, is located in the *`ArtixInstallDir`*`/artix/Version/schemas` directory and on the Internet. The references schema is needed whenever you generate WSDL from IDL that uses object references.

`-a <corba_address>` specifies a default value for the `location` attribute in the `corba:address` elements.

`-unwrap` generates doc/literal unwrapped style of WSDL.

`-usetypes` generates rpc/literal style of WSDL.

The default style of WSDL generated by the `idltowsdl` utility is doc/literal wrapped.

2. Edit the `corba:address` elements for each CORBA endpoint—for each CORBA endpoint, you have to specify the location of a CORBA object reference.

   Using your favorite text editor, open the `<IDLFile>.wsdl` file generated in the previous step. Replace the dummy setting, `location="..."`, in each of the `corba:address` elements, by one of the following location URL settings:

   - *File URL*—if the Orbix server writes an IOR to a file as it starts up, you specify the `location` attribute as follows:

     `location="file:///<DirPath>/<IORFile>.ior"`

     On Windows platforms, the URL format can indicate a particular drive—for example the `C:` drive—as follows:

     `location="file:///C:/<DirPath>/<IORFile>.ior"`

Note: It is usually simplest to specify the file name using an absolute path. If you specify the file name using a relative path, the location is taken to be relative to the directory the Artix process is started in, *not* relative to the containing WSDL file.

- *corbaname URL*—allows you to retrieve an object reference from the CORBA naming service. This setting has the following format:

  `location="corbaname:rir:/NameService#`*`StringName`*`"`

  Where *`StringName`* is a name in the CORBA naming service. For more details, see "How an Artix Client Resolves a Name" on page 58.
- *Stringified IOR*—if you know that the Orbix server's IOR is not going to change for some time, you can paste the stringified IOR directly into the location attribute, as follows:

  `location="IOR:000000..."`
- *Placeholder IOR*—is appropriate for IORs created dynamically at runtime (for example, IORs created by factory objects). In this case, you should use the special placeholder value, `IOR:`, for the location attribute, as follows:

  `location="IOR:"`

  Artix uses the enclosing `service` element as a template for transient object references.

For example, if your Orbix server writes an IOR to the file, `/tmp/app_iors/hello_world_service.ior`, you can use it to specify the endpoint location as follows:

```
<service name="HelloWorldCORBAService">
    <port binding="tns:HelloWorldCORBABinding" name="HelloWorldCORBAPort">
      <corba:address location="file:///tmp/app_iors/hello_world_service.ior"/>
    </port>
</service>
```

3. Generate SOAP bindings—generate a SOAP binding for each port type that you want to expose as a Web service. If you want to expose a single WSDL port type, enter the following command:

   ```
   > wsdltosoap -i <PortTypeName> -b <BindingName>
      <IDLFile>.wsdl
   ```

   Where *`<PortTypeName>`* refers to the `name` attribute of an existing `portType` element and *`<BindingName>`* is the name to be given to the newly generated SOAP binding. This command generates a new WSDL file, *`<IDLFile>`*`-soap.wsdl`.

If you want to expose *multiple* WSDL port types, you must run the `wsdltosoap` command iteratively, once for each port type. For example:

```
> wsdltosoap -i <PortType_A> -b <Binding_A>
   -o <IDLFile>01.wsdl <IDLFile>.wsdl
> wsdltosoap -i <PortType_B> -b <Binding_B>
   -o <IDLFile>02.wsdl <IDLFile>01.wsdl
> wsdltosoap -i <PortType_C> -b <Binding_C>
   -o <IDLFile>03.wsdl <IDLFile>02.wsdl
...
```

Where the `-o <FileName>` flag specifies the name of the output file. At the end of this step, rename the WSDL file to `router.wsdl`.

4. Add SOAP endpoints—add a `service` element for each of the port types you want to expose. For example, a simple SOAP endpoint could have the following form:

```
<definitions name="" targetNamespace="..."
    ...
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    ...>
    ...
    <service name="<SOAPServiceName>">
        <port binding="tns:<SOAPBinding>" name="<SOAPPortName>">
            <soap:address location="http://localhost:9000"/>
            <http-conf:client/>
            <http-conf:server/>
        </port>
    </service>
</definitions>
```

In the preceding example, you must add a line that defines the `http-conf` namespace prefix in the `<definitions>` tag.

The most important setting in the SOAP port is the `location` attribute of the `soap:address` element, which can be set to one of the following HTTP URLs:

- Explicit HTTP URL—if a particular service is meant to listen on a fixed address, you can specify the *<hostname>* and *<port>* values explicitly.

  `location="http://<hostname>:<port>`

- Placeholder HTTP URL—if a service is meant to be created dynamically at runtime, you should specify a transient HTTP URL, as follows:

  ```
  location="http://localhost:0
  ```

  At runtime, the placeholder URL is replaced by an explicit address when the service is created. Artix treats the enclosing `service` element as a template, allowing multiple transient services to be created at runtime.

Note: It is also possible to add a SOAP endpoint to the WSDL contract using the `wsdltoservice` command line tool. For details of this command, see the *Command Line Reference* document.

5. Add a route for each exposed port type—for each port type, you need to set up a route to translate incoming SOAP requests into outgoing CORBA requests. For example, the following route definition instructs the router to map incoming SOAP/HTTP request messages to a CORBA endpoint.

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    ...>
    ...
    <ns1:route name="route_0">
        <ns1:source      service="tns:<SOAPServiceName>"
                         port="<SOAPPortName>"/>
        <ns1:destination service="tns:<CORBAServiceName>"
                         port="<CORBAPortName>"/>
    </ns1:route>
</definitions>
```

In the preceding example, you must add a line that defines the `ns1` namespace prefix in the `<definitions>` tag.

The `ns1:source` element identifies the SOAP/HTTP endpoint in the router that receives incoming requests from a client. The

`ns1:destination` element identifies the CORBA endpoint in the Orbix server to which outgoing requests are routed.

Note: Generally, when defining routes, if the location of the source endpoint is a placeholder, the location of the destination endpoint should *also* be a placeholder.

6. Check that you have added all the namespaces that you need—for a typical SOAP/HTTP to CORBA route, you typically need to add the following namespaces (in addition to the namespaces already generated by default):

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...
    xmlns:tns="TargetNamespaceURI"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:references="http://schemas.iona.com/references"
    ...>
    ...
</definitions>
```

7. Include the references schema (if required)—if your IDL passes any object references (for example, as parameters or return values), the corresponding WSDL contract needs to include the references schema to represent the object references.

   For example, assuming that the `references.xsd` schema file is stored in the same directory as `router.wsdl`, you can include the references schema in the router contract as follows:

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...>
    <types>
        <schema targetNamespace="..."  ...>
            <import namespace="http://schemas.iona.com/references"
                schemaLocation="references.xsd"/>
            ...
        </schema>
    </types>
    ...
</definitions>
```

The original copy of the `references.xsd` schema file is located in the *`ArtixInstallDir`*`/artix/`*`Version`*`/schemas` directory.

router.wsdl file contents

For example, if the router contract contains a single port type, the contents of `router.wsdl` would have the following outline:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="" targetNamespace="TargetNamespaceURI"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
xmlns:corbatm="http://schemas.iona.com/typemap/corba/cdr_over_ii
   op.idl"
    xmlns:references="http://schemas.iona.com/references"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:http-conf="http://schemas.iona.com/transports/http/configu
   ration"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:tns="TargetNamespaceURI"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://schemas.iona.com/idltypes/cdr_over_iiop.idl">
    <types>
        ...
    </types>
    <message name="..."/>
    ...

    <portType name="<PortTypeName>">
        ...
    </portType>

    <binding name="<CORBABindingName>"
             type="tns:<PortTypeName>">
        ...
    </binding>

    <binding name="<SOAPBindingName>"
             type="tns:<PortTypeName>">
        ...
    </binding>

    <service name="<CORBAServiceName>">
        ...
    </service>

    <service name="<SOAPServiceName>">
```

```
        ...
    </service>

    <ns1:route name="route_0">
        <ns1:source      service="tns:<SOAPServiceName>"
                         port="<SOAPPortName>"/>
        <ns1:destination service="tns:<CORBAServiceName>"
                         port="<CORBAPortName>"/>
    </ns1:route>
</definitions>
```

Generate the client contract

The client WSDL contract is a modified copy of the router contract containing only those details of the contract that are relevant to the client. To generate the client contract, perform the following steps:

1. Copy the `router.wsdl` file to `client.wsdl`.
2. Edit the `client.wsdl` file to remove redundant elements. That is, you should remove the following:
   - CORBA `binding` elements.
   - CORBA `service` elements.
   - `route` elements.

   You could also optionally remove some of the redundant namespace definitions, such as `corba`, `corbatm`, and `ns1`.

client.wsdl file contents

For example, if the client contract contains a single port type, the contents of `client.wsdl` would have the following outline:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="" targetNamespace="TargetNamespaceURI"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
xmlns:corbatm="http://schemas.iona.com/typemap/corba/cdr_over_ii
   op.idl"
    xmlns:references="http://schemas.iona.com/references"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:http-conf="http://schemas.iona.com/transports/http/configu
   ration"
    xmlns:ns1="http://schemas.iona.com/routing"
    xmlns:tns="TargetNamespaceURI"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://schemas.iona.com/idltypes/cdr_over_iiop.idl">
```

```
    <types>
        ...
    </types>
    <message name="..."/>
    ...

    <portType name="<PortTypeName>">
        ...
    </portType>

    <binding name="<SOAPBindingName>"
             type="tns:<PortTypeName>">
        ...
    </binding>

    <service name="<SOAPServiceName>">
        ...
    </service>
</definitions>
```

Embedding Artix in a CORBA Service

Overview

If you want to expose an Orbix 6 CORBA server as a Web service, you have the option of embedding Artix directly in the CORBA server.

This embedding is possible because Artix and Orbix are both built using the same framework: IONA's Adaptive Runtime Technology (ART). Using the ART framework, it is possible to run Artix and Orbix in the same process just by loading the appropriate set of plug-ins needed by each product.

In this section

This section contains the following subsections:

Embedded Router Scenario

Overview

Figure 11 shows an overview of an Artix router embedded in a CORBA server. In this scenario, the CORBA service is exposed as a Web service that supports SOAP over HTTP. The embedded router is responsible for converting incoming SOAP/HTTP requests into colocated requests on the CORBA server. Any replies from the CORBA server are then converted into SOAP/HTTP replies by the router and sent back to the client.

Note: Embedding an Artix router is an option that is *only* available to Orbix 6 based CORBA applications. In general, the most straightforward way to build these applications is to use the Orbix libraries included with the Artix product. If you need to link with libraries taken directly from an Orbix distribution, you must take care to ensure that these libraries are binary compatible with Artix.

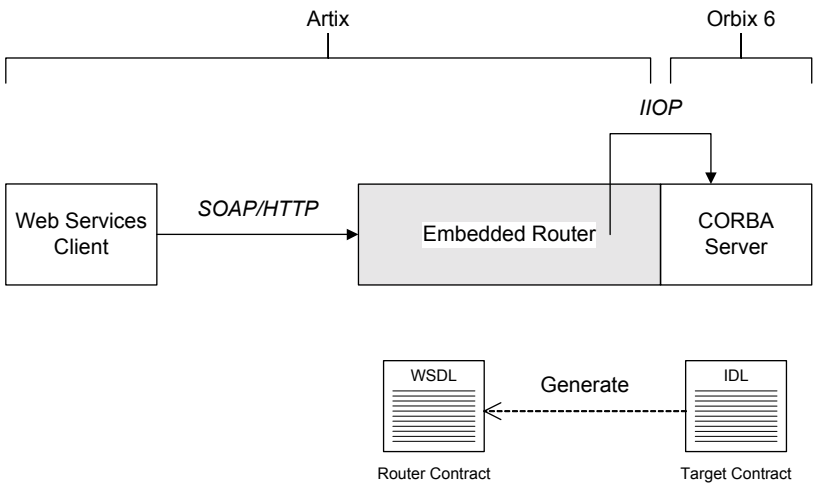

Figure 1: *Artix Router Embedded in a CORBA Server*

Modifications to CORBA server

The following changes must be made to the CORBA server to embed the Artix router:

- Code changes—*No*.
- Re-compilation—*No*.
- Configuration—modify the Orbix configuration file.

Elements required for this scenario

The following elements are required to implement this scenario:

- WSDL contract for clients.
- WSDL contract for the embedded router.
- Modified Orbix configuration file for the CORBA server.

Embedding a Router in the CORBA Server

Overview

This section describes how to embed a router in a CORBA server. The embedded router enables the CORBA server to receive requests from a SOAP/HTTP Web services client. The following steps are described:

- Convert IDL to WSDL.
- Deploy the requisite WSDL files.
- Edit the Artix configuration.

Convert IDL to WSDL

Use the Artix utilities to generate two WSDL files, `router.wsdl` and `client.wsdl`, from the CORBA server's IDL interface. For details of how to convert the IDL file to WSDL, see "Converting IDL to WSDL" on page 36.

Deploy the requisite WSDL files

Deploy the following WSDL files on the CORBA server host:

- `router.wsdl`—the router contract, which describes the route for converting SOAP/HTTP requests into CORBA requests.
- `references.xsd`—the schema that defines the `references:Reference` data type, which Artix uses to represent object references.

  The references schema is usually (but not always) required on the server side. If your IDL does not pass object references as parameters or return values, however, you do not need to deploy this file.

Edit the Artix configuration

Given that your CORBA server is configured by a particular configuration scope, `orbix_srvr_with_embeded_router`, Example 3 shows how to modify the server configuration to embed an Artix router.

Example 1: *Artix Configuration Suitable for an Embedded Artix Router*

```
# Artix Configuration File

orbix_srvr_with_embedded_router {
    ...

    # Modified configuration required for embedded router:
    #
```

Example 1: *Artix Configuration Suitable for an Embedded Artix Router*

```
1     orb_plugins = [..., "ws_orb", "soap", "at_http", "routing",
      "bus_loader"];
2     binding:client_binding_list = ["OTS+GIOP+IIOP", "GIOP+IIOP"];

3     plugins:routing:wsdl_url="../../etc/router.wsdl";

4     plugins:ws_orb:shlib_name = "it_ws_orb";
      plugins:soap:shlib_name = "it_soap";
      plugins:http:shlib_name = "it_http";
      plugins:at_http:shlib_name = "it_at_http";
      plugins:routing:shlib_name = "it_routing";
      plugins:bus_loader:shlib_name = "it_bus_loader";

5     share_variables_with_internal_orb = "false";
};
```

The preceding Artix configuration can be explained as follows:

1. Edit the ORB plug-ins, adding the requisite Artix plug-ins to the list. In this example, the following plug-ins are needed:
   - `ws_orb` plug-in—enables the router to send and receive CORBA messages.
   - `soap` plug-in—enables the router to send and receive SOAP messages.
   - `at_http` plug-in—enables the router to send and receive messages over the HTTP transport.
   - `routing` plug-in—contains the core of the Artix router.
   - `bus_loader` plug-in—triggers the Artix Bus initialization step. This plug-in is needed only when you are loading Artix plug-ins into a non-Artix application.

Note: In Artix 3.0, Artix plug-ins were refactored to cleanly separate the ORB initialization step from the Artix Bus initialization step. Usually, in an Artix application, `IT_Bus::init()` triggers the Bus initialization step. In this example, however, the CORBA server never calls `IT_Bus::init()`. Therefore, the `bus_loader` plug-in is needed to finish the initialization of the Artix plug-ins.

If you plan to use other bindings and transports, you might need to add some other Artix plug-ins instead.

2. The Artix embedded router is *not* compatible with the `POA_Coloc` interceptor. Therefore you must edit the server's `binding:client_binding_list` entry to remove any bindings containing the `POA_Coloc` interceptor.

   For example, if the client binding list is defined as follows:

   ```
   binding:client_binding_list =
      ["OTS+POA_Coloc","POA_Coloc","OTS+GIOP+IIOP","GIOP+IIOP"];
   ```

   You would replace it with the following list:

   ```
   binding:client_binding_list = ["OTS+GIOP+IIOP","GIOP+IIOP"];
   ```

   Note: If the `binding:client_binding_list` variable does not appear explicitly in the server's configuration scope, try to find it in the next enclosing scope (or the scope that is nearest to the server's configuration scope) and copy it into the server's scope.

   If you do not purge the `POA_Coloc` entries from the client binding list, clients that attempt to access the server through the router will receive a `CORBA::UNKNOWN` exception.

3. The `plugins:routing:wsdl_url` setting specifies the location of the router WSDL contract (see "Converting IDL to WSDL" on page 36). The URL can be a relative filename (as here) or a general `file:` URL.

4. In order for Orbix to load the Artix plug-ins, for each plug-in you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:<plugin_name>:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

   You can also specify additional configuration settings for the Artix plug-ins at this point (see the *Artix Configuration Reference* for more details).

5. In certain circumstances, Orbix creates an internal ORB instance (for example, during initialization). To prevent the settings from the current scope being used by the internal ORBs—specifically, to prevent the internal ORB from loading Artix plug-ins—you should set the `share_variables_with_internal_orb` configuration variable to `false`.

Exposing an Orbix 3.3 or Non-Orbix Service as a Web Service

Overview

If you want to expose an Orbix 3.3 or non-Orbix CORBA server as a Web service, it is generally necessary to deploy a standalone Artix router that acts as a bridge between Web services clients and the CORBA server. Using a standalone router is a non-intrusive integration approach that should work with any CORBA back-end.

In this section

This section contains the following subsections:

Standalone SOAP-to-CORBA Router Scenario

Overview

Figure 12 shows an overview of a standalone router. In this scenario, the router is packaged as a standalone application, which acts as a bridge between the Web services client and the CORBA server. The standalone router is responsible for converting incoming SOAP/HTTP requests into outgoing requests on the CORBA server. Replies from the CORBA server are converted into SOAP/HTTP replies by the router and sent back to the client.

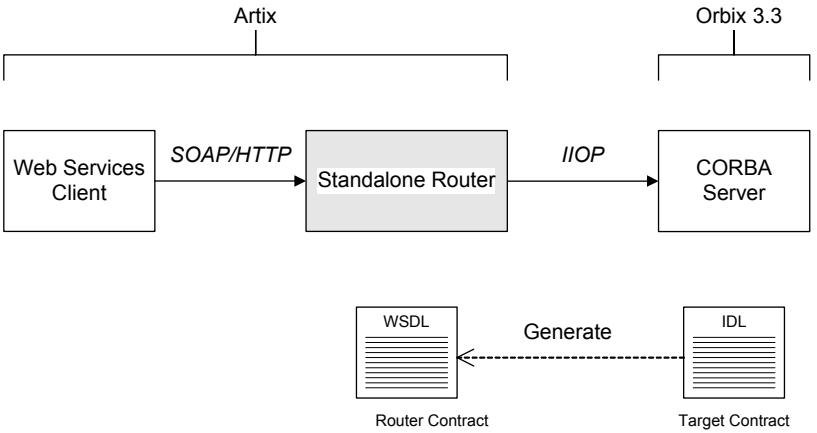

Figure 2: *Standalone Artix Router*

Container

The Artix container, `it_container`, is an application that can be used to run any of the standard Artix services. The functionality of the container is determined by the plug-ins it loads at runtime.

In this scenario, the container is configured to load the `router` plug-in (along with some other plug-ins) so that it functions as a standalone router.

Modifications to CORBA server

When using a standalone Artix router, no modifications need be made to the CORBA server.

Elements required for this scenario

The following elements are required to implement this scenario:

- WSDL contract for clients.
- WSDL contract for the standalone router.
- Artix configuration file for the standalone router.

Configuring and Running a Standalone SOAP-to-CORBA Router

Overview

This section describes how to configure and run a standalone router that acts as a bridge between a SOAP/HTTP Web services client and a CORBA server. The following steps are described:

- Convert IDL to WSDL.
- Deploy the requisite WSDL files.
- Create the Artix configuration.
- Run the standalone router.

Convert IDL to WSDL

Use the Artix utilities to generate two WSDL files, `router.wsdl` and `client.wsdl`, from the CORBA server's IDL interface. For details, see "Converting IDL to WSDL" on page 36.

Deploy the requisite WSDL files

Deploy the following WSDL files on the standalone router host:

- `router.wsdl`—the router contract, which describes the route for converting SOAP/HTTP requests into CORBA requests.
- `references.xsd`—the schema that defines the `references:Reference` data type, which Artix uses to represent object references.

  The references schema is usually (but not always) required on the server side. If your IDL does not pass object references as parameters or return values, however, you do not need to deploy this file.

Create the Artix configuration

Example 4 shows a suitable configuration for a standalone router that maps incoming SOAP/HTTP requests to outgoing CORBA requests.

Example 2: *Artix Configuration Suitable for a Standalone Artix Router*

```
# Artix Configuration File

standalone_router {
    # Configuration for standalone router:
    #
1   orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
    "iiop", "ws_orb", "soap", "at_http", "routing"];
```

Example 2: *Artix Configuration Suitable for a Standalone Artix Router*

```
    plugins:routing:wsdl_url="../../etc/router.wsdl";

    plugins:ws_orb:shlib_name = "it_ws_orb";
    plugins:soap:shlib_name = "it_soap";
    plugins:http:shlib_name = "it_http";
    plugins:at_http:shlib_name = "it_at_http";
    plugins:routing:shlib_name = "it_routing";

    # Uncomment these lines for interoperability with Orbix 3.3
    #policies:giop:interop_policy:negotiate_transmission_codeset
    = "false";
    #policies:giop:interop_policy:send_principal = "true";
    #policies:giop:interop_policy:send_locate_request = "false";
};
```

The preceding Artix configuration can be explained as follows:

1. Edit the ORB plug-ins, adding the requisite Artix plug-ins to the list. In this example, the following plug-ins are needed:
   - `xmlfile_log_stream` plug-in—enables logging to an XML file.
   - `iiop_profile`, `giop`, and `iiop` plug-ins—enables the IIOP protocol (used by CORBA).
   - `ws_orb` plug-in—enables the router to send and receive CORBA messages.
   - `soap` plug-in—enables the router to send and receive SOAP messages.
   - `at_http` plug-in—enables the router to send and receive messages over the HTTP transport.
   - `routing` plug-in—contains the core of the Artix router.

   If you plan to use other bindings and transports, you might need to add some other Artix plug-ins instead.
2. The `plugins:routing:wsdl_url` setting specifies the location of the router WSDL contract (see "Converting IDL to WSDL" on page 36). The URL can be a relative filename (as here) or a general `file:` URL.
3. To load the Artix plug-ins, you must specify the *root name* of the shared library (or DLL on Windows) that contains the plug-in code. The requisite `plugins:<plugin_name>:shlib_name` entries can be copied from the root scope of the Artix configuration file, `artix.cfg`.

You can also specify additional plug-in configuration settings at this point (see the *Artix Configuration Reference* for more details).

4. If the router needs to integrate with an Orbix 3.3 CORBA server, you should uncomment these lines to enable interoperability. For more details about these configuration settings, see the *Artix Configuration Reference*.

> **Note:** These interoperability settings might also be useful for integrating with other third-party ORB products. See the *Artix Configuration Reference* for more details.

Run the standalone router

Run the standalone router by invoking the container, `it_container`, passing the router's ORB name as a command-line parameter (the ORB name is identical to the name of the router's configuration scope).

For example, to run the router configured in Example 4 on page 54, enter the following at a command prompt:

```
it_container -ORBname standalone_router
```

CHAPTER 4

Integrating the CORBA Naming Service with Artix

In a mixed Artix/CORBA system, it is often necessary for an Artix application to retrieve an object reference from the CORBA Naming Service. Artix supports a relatively simple configuration option for binding a name to or resolving a name from the CORBA Naming Service: simply set the location attribute of <corba:address> to be a corbaname URL.

In this chapter

This chapter discusses the following topics:

How an Artix Client Resolves a Name

Overview

Figure 13 shows a typical scenario where an Artix client might need to resolve a name from the CORBA Naming Service. The Artix client, which is configured to have a `corba` binding, connects to a pure CORBA server using the CORBA Naming Service.

To configure the client to resolve the name, you need to specify a `corbaname` URL in the `corba:address` element within a `service`. No programming is required. There are, however, some prerequisites settings in the Artix configuration file that are also required in order to enable the client to find the CORBA Naming Service.

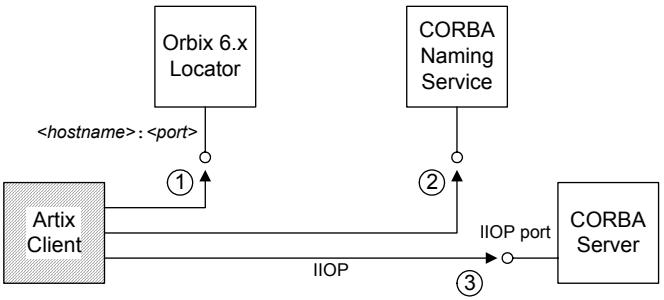

Figure 1: *Artix Client Resolving a Name from the Naming Service*

Resolving steps for Orbix 6.x

Artix performs the following steps to resolve a name in the Orbix 6.x CORBA Naming Service (as shown in Figure 13):

Step	Action
1	The Artix client sends a GIOP *LocateRequest* message to the Orbix locator, whose hostname and port is specified in the Artix configuration file. The *LocateRequest* reply gives the location of the CORBA Naming Service.
2	The Artix client contacts the CORBA Naming Service to resolve the name specified in the WSDL `corba:address` element.
3	The object reference returned from the naming service is used to contact the CORBA server.

Prerequisites

Before configuring the client's WSDL contract to resolve a name from the CORBA Naming Service, you must edit the Artix configuration file to provide some details about the remote naming service. The configuration settings depend on the kind of ORB you are interoperating with, as follows:

Interoperating with Orbix 6.x, ASP 5.x

In your Artix configuration file, *ArtixInstallDir*`/artix/`*Version*`/etc/domains/artix.cfg`, add the following lines to the configuration scope used by the Artix client:

```
# Artix Configuration File
artix_client_of_Orbix_6 {
    ...
    initial_references:NameService:reference = "corbaloc::<hostname>:<port>/NameService";
    url_resolvers:corbaname:plugin="naming_resolver";
    plugins:naming_resolver:shlib_name="it_naming";
};
```

Where *<hostname>*:*<port>* is the host and port where the Orbix locator service is running. By default, Orbix 6.x configures the locator *<port>* to be 3075, but you might need to check the `plugins:locator:iiop:port` setting in your Orbix 6.x configuration file if you are not sure of the value.

Note: The *Orbix locator service* is responsible for keeping track of running Orbix services. It is completely unrelated to the Artix locator service.

Interoperating with Orbix 3.3

In your Artix configuration file, *ArtixInstallDir*/artix/*Version*/etc/domains/artix.cfg, add the following lines to the configuration scope used by the Artix client:

```
# Artix Configuration File
artix_client_or_Orbix_33 {
    ...
    initial_references:NameService:reference = "IOR:000000......";

    policies:giop:interop_policy:negotiate_transmission_codeset = "false";
    policies:giop:interop_policy:send_principal = "true";
    policies:giop:interop_policy:send_locate_request = "false";
};
```

The stringified IOR shown in the preceding example, `IOR:000000...`, can be obtained from the 3.3.x Naming Service by starting the NS with the `-I` *<filename>* switch and copying the IOR from the *<filename>* into the configuration file. When using the `IOR:` format, you do not need to load the `naming_resolver` plug-in (the `naming_resolver` is needed only to resolve `corbaloc` URLs).

Interoperating with other ORBs

Generally, the approach used for interoperating with Orbix 3.3 (initializing `initial_references:NameService:reference` with the value of the naming service's IOR) should work for just about any third-party ORB product. You might need to modify some of the GIOP interoperability policies, however. For more details, consult the *Artix Configuration Reference*.

Configure the WSDL service

To configure an Artix client to resolve a name in the CORBA Naming Service, use the `corbaname` URL format in the `<corba:address>` tag, as follows:

```
<service name="CORBAService">
    <port binding="tns:CORBABinding" name="CORBAPort">
        <corba:address location="corbaname:rir:/NameService#StringName"/>
    </port>
</service>
```

Where *`StringName`* is the name that you want to resolve, specified in the standard CORBA Naming Service string format. For example, if you have a name with `id` equal to `ArtixTest` and `kind` equal to `obj`, contained within a naming context with `id` equal to `Foo` and `kind` equal to `ctx`, the `corbaname` URL would be expressed as:

```
corbaname:rir:/NameService#Foo.ctx/ArtixTest.obj
```

In other words, the general format of a string name is as follows:

`<id>[.<kind>]/<id>[.<kind>]/...`

How an Artix Server Binds a Name

Overview

Figure 14 shows a typical scenario where an Artix server might need to bind a name to the CORBA Naming Service. In the context of the CORBA Naming Service, *binding a name* means that the server advertises the location of a CORBA object by storing an object reference against a name in the naming service.

To configure the server to bind the name, you need to specify a `corbaname` URL in the `corba:address` element within a `service` (exactly the same configuration as an Artix client). When the Artix server activates the `<service>` or `<port>`, by registering with the Artix Bus, the runtime automatically binds the name in the naming service.

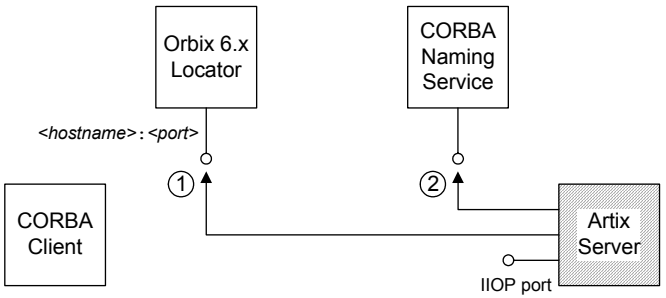

Figure 2: *Artix Server Binding a Name to the Naming Service*

Binding steps for Orbix 6.x

Artix performs the following steps to bind a name in the Orbix 6.x CORBA Naming Service (as shown in Figure 14):

Step	Action
1	The Artix server sends a GIOP *LocateRequest* message to the Orbix locator, whose hostname and port is specified in the Artix configuration file. The *LocateRequest* reply gives the location of the CORBA Naming Service.
2	The Artix server contacts the CORBA Naming Service to bind the name specified in the WSDL `corba:address` element.

Prerequisites

The prerequisites for an Artix server that binds a name to the CORBA Naming Service are identical to the prerequisites for an Artix client that resolves a name—see "Prerequisites" on page 59 for details.

Configure the WSDL service

To configure an Artix server to bind a name in the CORBA Naming Service, use the `corbaname` URL format in the `<corba:address>` tag, as follows:

```
<service name="CORBAService">
    <port binding="tns:CORBABinding" name="CORBAPort">
        <corba:address location="corbaname:rir:/NameService#StringName"/>
    </port>
</service>
```

Where *`StringName`* is the name that you want to resolve, specified in the standard CORBA Naming Service string format.

This is identical to the configuration for an Artix client, but the server treats this configuration setting differently. When an Artix server activates a service containing a `corbaname` URL, the server automatically binds the given *`StringName`* into the CORBA naming service.

Binding semantics

The automatic binding performed by an Artix server when it encounters a `corbaname` URL has the following characteristics:

- The binding operation has the semantics of the `CosNaming::NamingContext::rebind()` IDL operation. That is, the bind operation either creates a new binding or clobbers an existing binding of the same name.
- If some of the naming contexts in the *`StringName`* compound name do not yet exist in the naming service, the Artix server does *not* create the missing contexts.

  For example, if you try to bind a *`StringName`* with the value `Foo/Bar/SomeName` where neither the `Foo` nor `Foo/Bar` naming contexts exist yet, the Artix server will not bind the given name. You would need to create the naming contexts manually prior to running the Artix server (for example, in Orbix 6.x you could issue the command `itadmin ns newnc` *`NameContext`*).

Artix Client Integrated with a CORBA Server

Overview

This section presents an example scenario of an Artix client integrated with a CORBA server, where the client obtains a CORBA object reference through the CORBA Naming Service.

In summary, the scenario works as follows:

- A CORBA Naming Service from an ORB product (presumed to be Orbix 6.x) is assumed to be running.
- As the CORBA server starts up, it uses the `CosNaming::NamingContext` IDL interface to bind a name to the naming service.
- When the Artix client starts up, the Artix runtime reads the client's WSDL contract, extracts a `corbaname` URL and contacts the naming service to resolve the `corbaname` URL.

In this section

This section contains the following subsections:

CORBA Server Implementation

Overview

The code example in this subsection shows you how a server binds a name to the root naming context of the CORBA Naming Service. This shows how a CORBA programmer can use the standard `CosNaming::NamingContext` IDL interface to bind a name.

Note: This is a pure CORBA example; there is no Artix programming involved here.

CORBA server main function

Example 5 shows part of the `main()` function for a CORBA server that registers a name in the CORBA Naming Service. The lines of code shown in bold bind the name, `ArtixTest`, to the root naming context.

Example 1: *CORBA Server that Register a Name in the Naming Service*

```
// C++
...
#include <omg/CosNaming.hh>
...
int main(int argc, char* argv[])
{
    IT_TerminationHandler::set_signal_handler(sig_handler);

    try
    {
        cout << "Initializing the ORB" << endl;
        global_orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var poa_obj =
            global_orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var root_poa =
            PortableServer::POA::_narrow(poa_obj);
        assert(!CORBA::is_nil(root_poa));

        cout << "Creating objects" << endl;

        HWImplementation hw_servant;
        PortableServer::ObjectId_var hw_oid =
            root_poa->activate_object(&hw_servant);
```

Example 1: *CORBA Server that Register a Name in the Naming Service*

```
        CORBA::Object_var ref=
    root_poa->create_reference_with_id(
                                    hw_oid,
                                    _tc_HelloWorld->id()
                                );

        // Use the simple NamingContext interface
        CosNaming::NamingContext_var rootContext;

        // Get a reference to the Root Naming Context.
        CORBA::Object_var objVar;
        objVar = global_orb->resolve_initial_references(
                                    "NameService"
                                );
        rootContext = CosNaming::NamingContext::_narrow(objVar);

        if (CORBA::is_nil(rootContext.in()))
        {
            cerr << "_narrow returned nil" << endl;
            return 1;
        }

        CosNaming::Name_var tmpName = new CosNaming::Name(1);
        tmpName->length(1);

        tmpName[0].id = CORBA::string_dup("ArtixTest");
        tmpName[0].kind = CORBA::string_dup("");
        rootContext->rebind(tmpName, ref);

        // Activate the POA Manager to allow requests to arrive
        PortableServer::POAManager_var poa_manager =
                                        root_poa->the_POAManager();
        poa_manager->activate();

        // Give control to the ORB
        //
        global_orb->run();
        return 0;
    }
    catch (CORBA::Exception& e)
    {
        cout << "Error occurred: " << e << endl;
    }
    return 1;
}
```

Demonstration code

If you want to run this CORBA server code in a real example, you could use the following demonstration as a starting point:

ArtixInstallDir/artix/*Version*/demos/transports/cdr_over_iiop/corba

In the server subdirectory, there is an existing `server.cxx` mainline file that publishes the IOR by saving to a file. To change the server to use the naming service, you can replace the existing server `main()` function with the code shown in Example 5 on page 66.

Note the following points:

- Remember to add the include line, `#include <omg/CosNaming.hh>`, at the start of the `server.cxx` file.
- Edit the server `Makefile`, adding the `it_naming` library to the link list. For example, on Windows you would add `it_naming.lib` to the link list.
- You need a separate ORB product (for example, Orbix) to run the CORBA Naming Service. The Artix product does *not* include a CORBA Naming Service.

Artix Client Configuration

Overview

This subsection shows how to configure an Artix client to fetch an object reference from the CORBA Naming Service.

Demonstration configuration

The configuration files referred to in this subsection are taken from the `cdr_over_iiop` demonstration and located in the following directory:

ArtixInstallDir`/artix/`*Version*`/demos/transports/cdr_over_iiop/etc`

The corresponding client application requires no modification. You can choose to run either a C++ version of the client:

`cdr_over_iiop/cxx/client`

Or a Java version of the client:

`cdr_over_iiop/java/client`

Artix configuration file

Example 6 shows the Artix configuration required for the Artix client to interoperate with the Orbix 6.x naming service.

Example 2: *Artix Configuration for Interoperating with Orbix 6 Naming*

```
# Artix Configuration File
include "../../../../../etc/domains/artix.cfg";

demos {
  cdr_over_iiop {
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop", "iiop", "ws_orb"];

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

    corba {
      orb_plugins = ["iiop_profile", "giop", "iiop"];
    };
  };
};
```

To configure the `cdr_over_iiop` demonstration, edit the `cdr_over_iiop/etc/cdr_over_iiop.cfg` file, inserting the three lines highlighted in bold in Example 6 on page 69. You might need to modify the

value of the hostname and port—this example assumes that the Orbix locator service is running on the same host as the client, `localhost`, and listening on the default port, `3075`.

Note: The configuration shown in Example 6 on page 69 is specific to the Orbix 6.x naming service. If you use a different ORB product, you might have to set this configuration differently—see "Prerequisites" on page 59 for more details.

WSDL contract

You also need to edit the client's WSDL contract, specifying the `location` attribute of the `corba:address` element using a `corbaname` URL. Example 7 shows the modifications you need to make to the `corba:address` element in the `cdr_over_iiop/etc/cdr_over_iiop.wsdl` contract file.

Example 3: *CORBA Address Specified as a corbaname URL*

```
<definitions name="cdr_over_iiop" targetNamespace="http://www.iona.com/cdr_over_iiop"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    xmlns:corbatm="http://www.iona.com/cdr_over_iiop"
    ... >
    ...
    <service name="HelloWorldService">
        <port binding="corbatm:HelloWorldBinding" name="HelloWorldPort">
            <corba:address location="corbaname:rir:/NameService#ArtixTest"/>
        </port>
    </service>
</definitions>
```

When the client starts up, the Artix runtime automatically retrieves the CORBA object reference by resolving the name, `ArtixTest`, in the scope of the root naming context.

CHAPTER 5

Advanced CORBA Port Configuration

This chapter describes some advanced configuration options for customizing a CORBA port on an Artix server.

In this chapter

This chapter discusses the following topics:

Configuring Fixed Ports and Long-Lived IORs

Overview

Artix provides a `corba:policy` element that enables you to customize certain CORBA-specific policies for a WSDL service that acts as a CORBA endpoint. Essentially, the `corba:policy` element makes it possible to enable the following features on a CORBA endpoint:

- *Fixed IP port*—the WSDL service listens on the same IP port all the time. This is useful, for example, if the available range of IP ports is restricted or if the service must be accessible through a firewall.
- *Long-lived interoperable object references (IORs)*—the IOR remains valid even after the server is stopped and restarted.

You can configure a WSDL service to behave in one of the following ways:

- Transient service.
- Direct persistent service.

Transient service

By default, a CORBA endpoint is automatically configured to be *transient*. A transient service generates IORs with the following characteristics:

- *Randomly-assigned IP port*—the IP port is assigned by the underlying operating system. Hence, the port is generally different each time the Artix server is run.
- *Short-lived IORs*—the CORBA binding generates IORs in such a way that they are guaranteed to become invalid when the server is stopped and restarted.

Note: In this context, *transient* is a CORBA concept which refers to the `TRANSIENT` value of the `PortableServer::LifespanPolicy`. This notion of transience should *not* be confused with the Artix notion of transience, which is concerned with registering transient servants. The two concepts are completely different.

Direct persistent service

You can optionally configure a CORBA endpoint to be *direct persistent*. A direct persistent service generates IORs with the following characteristics:

- *Fixed IP port*—you can explicitly assign the IP port by configuration. Hence, the IP port remains the same each time the Artix server is run.
- *Long-lived IORs*—the CORBA binding generates IORs in such a way that they remain valid even when the server is stopped and restarted. All of the addressing information embedded in the IOR must remain constant, in particular:
  - *IP port is fixed*—the WSDL service must be configured to listen on a fixed IP port.
  - *POA name is fixed*—the POA name is a CORBA-specific construct that identifies an endpoint.
  - *Object ID in IOR is fixed*—the Object ID is a CORBA-specific construct that identifies a particular object in a given POA instance.
  - *POA is persistent*—a prerequisite for generating long-lived IORs is that the POA must have a life span policy value of `PERSISTENT`.

Configuring a service to be direct persistent

To configure an Artix service to be direct persistent, you must edit both the WSDL file and the Artix configuration file.

Editing the WSDL file

Artix enables you to set direct persistence attributes in WSDL by adding a `corba:policy` element to the WSDL service, as shown in Example 8.

Example 1: *Setting Direct Persistence Attributes in WSDL*

```
<definitions name="" targetNamespace="..."
    ...
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    ...>
    ...
    <service name="CORBAServiceName">
        <port binding="tns:CORBABinding" name="CORBAPortName">
            <corba:address location="file:///greeter.ior"/>
            <corba:policy persistent="true"
                          poaname="FQPN"
                          serviceid="ObjectID" />
        </port>
    </service>
</definitions>
```

The `corba:policy` attributes from Example 8 can be explained as follows:

- `persistent` attribute—by setting this attribute to `true`, you configure the CORBA binding to generate persistent IORs (that is, IORs that continue to be valid even after the Artix server is stopped and restarted). The default value is `false`.

  Note: In CORBA terms, this is equivalent to setting the `PortableServer::LifespanPolicy` policy to `PERSISTENT`.

- `poaname` attribute—in CORBA terminology, a POA is an object that groups CORBA objects together (a kind of container for CORBA objects). It is necessary to set the POA name here, because the POA name is embedded in the generated IORs. The generated IORs would

not be long-lived, unless the POA name remains constant. By default, a POA name is automatically generated with the value, `{ServiceNamespace}ServiceLocalPart#PortName`.

Note: The POA name, *FQPN*, is a *fully-qualified POA name*. In practice, however, you can only set a simple POA name. Artix currently does not provide a way of creating a POA name hierarchy.

- `serviceid` attribute—in CORBA terminology, this attribute specifies an *Object ID* for a CORBA object. It is necessary to set the Object ID here, because the Object ID is embedded in the server-generated IOR. The Object ID must have a constant value in order for the IOR to be long-lived. By default, the underlying POA would generate a random value for the Object ID.

  Artix currently allows you to set only one Object ID for each port.

  Note: The `serviceid` attribute also implicitly sets the `PortableServer::IdAssignmentPolicy` policy to `USER_ID`. If the `serviceid` attribute is *not* set, the `PortableServer::IdAssignmentPolicy` policy defaults to `SYSTEM_ID`.

Editing the Artix configuration file

To complete the configuration of direct persistence, you must also set some configuration variables in the relevant scope of the Artix configuration file. For example, if your Artix server uses the `artix_server` configuration scope, you would add the configuration variables as shown in Example 9.

Example 2: *Setting Direct Persistence Configuration Variables*

```
# Artix Configuration File
...
artix_server {
    ...
    poa:FQPN:direct_persistent="true";
    poa:FQPN:well_known_address="WKA_prefix";
    WKA_prefix:iiop:port="IP_Port";
};
```

The configuration variables from Example 9 can be explained as follows:

- `poa:`*`FQPN`*`:direct_persistent` variable—you must set this variable to `true`, which configures the CORBA binding to receive *direct* connections from Orbix clients. You should substitute *`FQPN`* with the POA name from the `poaname` attribute in the WSDL (see Example 8 on page 74).

  Note: In CORBA terms, this is equivalent to setting the `IT_PortableServer::PersistenceModePolicy` policy to `DIRECT_PERSISTENCE`. The alternative policy value, `INDIRECT_PERSISTENCE`, is not compatible with Artix, because it would require connections to be routed through the *Orbix locator service*, which is *not* part of the Artix product.

- `poa:`*`FQPN`*`:well_known_address` variable—this variable defines a prefix, *`WKA_prefix`*, which forms part of the variable names that configure a fixed port for the WSDL service. You should substitute *`FQPN`* with the POA name from the `poaname` attribute in the WSDL.
- *`WKA_prefix`*`:iiop:port` variable—this variable configures a fixed IP port for the WSDL service associated with *`WKA_prefix`*.

Fixed port configuration variables

The following IIOP configuration variables can be set for a CORBA endpoint that uses the *WKA_prefix* prefix:

`WKA_prefix``:iiop:host = "`*`host`*`";`

Specifies the hostname, *`host`*, to publish in the IIOP profile of server-generated IORs. This variable is potentially useful for multi-homed hosts, because it enables you to specify which network card the client should attempt to connect to.

`WKA_prefix``:iiop:port = "`*`port`*`";`

Specifies the fixed IP port, *`port`*, on which the server listens for incoming IIOP/TLS messages. This port value is also published in the IIOP profile of generated IORs.

`WKA_prefix``:iiop:listen_addr = "`*`host`*`";`

Restricts the IIOP/TLS listening point to listen only on the specified address, *`host`*. It is generally used on multi-homed hosts to limit incoming connections to a particular network interface. The default is to listen on `0.0.0.0` (which represents every network card on the host).

Secure fixed port configuration variables

Additionally, the following secure fixed port configuration variables can be set for a CORBA endpoint that uses the *WKA_prefix* prefix:

```
WKA_prefix:iiop_tls:host
WKA_prefix:iiop_tls:port
WKA_prefix:iiop_tls:listen_addr
```

These configuration variables function analogously to their insecure counterparts.

Note: These secure configuration variables will have no effect, unless the `iiop_tls` plug-in is also loaded. It is strongly recommended that you read the *Artix Security Guide* for details of how to configure IIOP/TLS security.

CORBA Timeout Policies

Overview

Artix servers that expose a CORBA endpoint can be configured to use CORBA-specific timeout policies. The timeout policies described here affect GIOP transports (for example, the IIOP or IIOP/TLS transports), but do *not* have any affect on non-CORBA transports.

Example

To use the timeout policies, add the relevant configuration variables to the Artix server's configuration scope in the Artix configuration file. For example, for an Artix server that uses the `artix_server` configuration scope, you can set the CORBA relative roundtrip timeout as follows:

```
# Artix Configuration File
artix_server {
    # Limit total time for an invocation to 2 seconds
    # (including time for connection and binding establishment).
    policies:relative_roundtrip_timeout = "2000";
}
```

Timeout policies

You can configure the following CORBA timeout policies in your Artix configuration file:

`policies:relative_binding_exclusive_request_timeout`
: Limits the amount of time allowed to deliver a request, exclusive of binding attempts. Request delivery is considered complete when the last fragment of the GIOP request is sent over the wire to the target object. This policy's value is set in millisecond units.

`policies:relative_binding_exclusive_roundtrip_timeout`
: Limits the amount of time allowed to deliver a request and receive its reply, exclusive of binding attempts. The countdown begins immediately after a binding is obtained for the invocation. This policy's value is set in millisecond units.

`policies:relative_connection_creation_timeout`
: Specifies how much time is allowed to resolve each address in an IOR, within each binding iteration. Defaults to 8 seconds.

An IOR can have several `TAG_INTERNET_IOP` (IIOP transport) profiles, each with one or more addresses, while each address can resolve through DNS to multiple IP addresses.

This policy applies to each IP address within an IOR. Each attempt to resolve an IP address is regarded as a separate attempt to create a connection. The policy's value is set in millisecond units.

`policies:relative_request_timeout`

Specifies how much time is allowed to deliver a request. Request delivery is considered complete when the last fragment of the GIOP request is sent over the wire to the target object. The timeout-specified period includes any delay in establishing a binding. This policy type is useful to a client that only needs to limit request delivery time. Set this policy's value in millisecond units.

No default is set for this policy; if it is not set, request delivery has unlimited time to complete.

`policies:relative_roundtrip_timeout`

Specifies how much time is allowed to deliver a request and its reply. Set this policy's value in millisecond units. No default is set for this policy; if it is not set, a request has unlimited time to complete.

The timeout countdown begins with the request invocation, and includes the following activities:

- Marshalling in/inout parameters
- Any delay in transparently establishing a binding

If the request times out before the client receives the last fragment of reply data, all received reply data is discarded. In some cases, the client might attempt to cancel the request by sending a GIOP `CancelRequest` message.

Retrying Invocations and Rebinding

Overview

Artix lets you configure CORBA policies that customize invocation retries and reconnection. The policies can be grouped into the following categories:

- Retrying invocations.
- Rebinding.

Retrying invocations

The following configuration variables determine how the CORBA binding deals with requests that raise the `CORBA::TRANSIENT` exception with a completion status of `COMPLETED_NO`. In terms of an IIOP connection, a `TRANSIENT` exception is raised if an error occurred before or during an attempt to write to or connect to a socket.

`policies:invocation_retry:backoff_ratio`
: Specifies the degree to which delays between invocation retries increase from one retry to the next. Defaults to 2.

`policies:invocation_retry:initial_retry_delay`
: Specifies the amount of time, in milliseconds, between the first and second retries. Defaults to 100.

> **Note:** The delay between the initial invocation and first retry is always `0`.

`policies:invocation_retry:max_forwards`
: Specifies the number of times an invocation message can be forwarded. Defaults to 20. To specify unlimited forwards, set to -1.

`policies:invocation_retry:max_retries`
: Specifies the number of transparent reinvocations attempted on receipt of a `TRANSIENT` exception. Defaults to 5.

Rebinding

The following configuration variables determine how the CORBA binding deals with requests that raise the `CORBA::COMM_FAILURE` exception with a completion status of `COMPLETED_NO`. In terms of an IIOP connection, a `COMM_FAILURE` exception is raised with a completion status of `COMPLETED_NO`, if the connection went down.

`policies:rebind_policy`

Specifies the default value for the rebind policy. Can be one of the following:

- `TRANSPARENT` *(default)*
- `NO_REBIND`
- `NO_RECONNECT`

`policies:invocation_retry:max_rebinds`

Specifies the number of transparent rebinds attempted on receipt of a `COMM_FAILURE` exception. Defaults to 5.

> **Note:** This setting is valid only if the effective `policies:rebind_policy` value is `TRANSPARENT`; otherwise, no rebinding occurs.

CHAPTER 6

Artix IDL to C++ Mapping

This chapter describes how Artix maps IDL to C++; that is, the mapping that arises by converting IDL to WSDL (using the IDL-to-WSDL compiler) and then WSDL to C++ (using the WSDL-to-C++ compiler).

In this chapter

This chapter discusses the following topics:

Introduction to IDL Mapping

Overview

This chapter gives an overview of the Artix IDL-to-C++ mapping. Mapping IDL to C++ in Artix is performed as a two step process, as follows:

1. Map the IDL to WSDL using the Artix IDL compiler. For example, you could map a file, `SampleIDL.idl`, to a WSDL contract, `SampleIDL.wsdl`, using the following command:

   ```
   idl -wsdl SampleIDL.idl
   ```

2. Map the generated WSDL contract to C++ using the WSDL-to-C++ compiler. For example, you could generate C++ stub code from the `SampleIDL.wsdl` file using the following command:

   ```
   wsdltocpp SampleIDL.wsdl
   ```

For a detailed discussion of these command-line utilities, see the *Artix User's Guide*.

Alternative C++ mappings

If you are already familiar with CORBA technology, you will know that there is an existing standard for mapping IDL to C++ directly, which is defined by the Object Management Group (OMG). Hence, two alternatives exist for mapping IDL to C++, as follows:

- Artix IDL-to-C++ mapping—this is a two stage mapping, consisting of IDL-to-WSDL and WSDL-to-C++. It is an IONA-proprietary mapping.
- CORBA IDL-to-C++ mapping—as specified in the OMG C++ Language Mapping document (http://www.omg.org). This mapping is used, for example, by the IONA's Orbix.

These alternative approaches are illustrated in Figure 15.

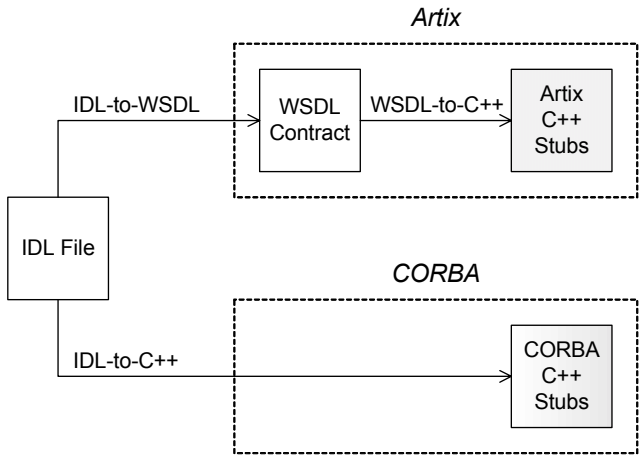

Figure 1: *Artix and CORBA Alternatives for IDL to C++ Mapping*

The advantage of using the Artix IDL-to-C++ mapping in an application is that it removes the CORBA dependency from your source code. For example, a server that implements an IDL interface using the Artix IDL-to-C++ mapping can also interoperate with other Web service protocols, such as SOAP over HTTP.

Unsupported IDL types

The following IDL types are not supported by the Artix C++ mapping:

- `wchar`.
- `wstring`.
- `long double`.
- Value types.
- Boxed values.
- Local interfaces.
- Abstract interfaces.
- forward-declared interfaces.

IDL Basic Type Mapping

Overview

Table 1 shows how IDL basic types are mapped to WSDL and then to C++.

Table 1: *Artix Mapping of IDL Basic Types to C++*

IDL Type	WSDL Schema Type	C++ Type
any	xsd:anyType	IT_Bus::AnyHolder
boolean	xsd:boolean	IT_Bus::Boolean
char	xsd:byte	IT_Bus::Byte
string	xsd:string	IT_Bus::String
wchar	xsd:string	IT_Bus::String
wstring	xsd:string	IT_Bus::String
short	xsd:short	IT_Bus::Short
long	xsd:int	IT_Bus::Int
long long	xsd:long	IT_Bus::Long
unsigned short	xsd:unsignedShort	IT_Bus::UShort
unsigned long	xsd:unsignedInt	IT_Bus::UInt
unsigned long long	xsd:unsignedLong	IT_Bus::ULong
float	xsd:float	IT_Bus::Float
double	xsd:double	IT_Bus::Double
long double	*Not supported*	*Not supported*
octet	xsd:unsignedByte	IT_Bus::UByte
fixed	xsd:decimal	IT_Bus::Decimal
Object	references:Reference	IT_Bus::Reference

Mapping for string

The IDL-to-WSDL mapping for strings is ambiguous, because the `string`, `wchar`, and `wstring` IDL types all map to the same type, `xsd:string`. This ambiguity can be resolved, however, because the generated WSDL records the original IDL type in the CORBA binding description (that is, within the scope of the `<wsdl:binding> </wsdl:binding>` tags). Hence, whenever an `xsd:string` is sent over a CORBA binding, it is automatically converted back to the original IDL type (`string`, `wchar`, or `wstring`).

IDL Complex Type Mapping

Overview

This section describes how the following IDL data types are mapped to WSDL and then to C++:

- enum type.
- struct type.
- union type.
- sequence types.
- array types.
- exception types.
- typedef of a simple type.
- typedef of a complex type.

enum type

Consider the following definition of an IDL enum type, `SampleTypes::Shape`:

```
// IDL
module SampleTypes {
    enum Shape { Square, Circle, Triangle };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Shape` enum to a WSDL restricted simple type, `SampleTypes.Shape`, as follows:

```
<xsd:simpleType name="SampleTypes.Shape">
   <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Square"/>
       <xsd:enumeration value="Circle"/>
       <xsd:enumeration value="Triangle"/>
   </xsd:restriction>
</xsd:simpleType>
```

The WSDL-to-C++ compiler maps the `SampleTypes.Shape` type to a C++ class, `SampleTypes_Shape`, as follows:

```
class SampleTypes_Shape : public IT_Bus::AnySimpleType
{
  public:
    SampleTypes_Shape();
    SampleTypes_Shape(const IT_Bus::String & value);
    ...
    void set_value(const IT_Bus::String & value);
    const IT_Bus::String & get_value() const;
};
```

The value of the enumeration type can be accessed and modified using the `get_value()` and `set_value()` member functions.

---

union type

Consider the following definition of an IDL union type, `SampleTypes::Poly`:

```
// IDL
module SampleTypes {
    union Poly switch(short) {
        case 1: short theShort;
        case 2: string theString;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Poly` union to an XML schema choice complex type, `SampleTypes.Poly`, as follows:

```
<xsd:complexType name="SampleTypes.Poly">
    <xsd:choice>
        <xsd:element name="theShort" type="xsd:short"/>
        <xsd:element name="theString" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

The WSDL-to-C++ compiler maps the `SampleTypes.Poly` type to a C++ class, `SampleTypes_Poly`, as follows:

```
// C++
class SampleTypes_Poly : public IT_Bus::ChoiceComplexType
{
  public:
    ...
    const IT_Bus::Short gettheShort() const;
    void settheShort(const IT_Bus::Short& val);

    const IT_Bus::String& gettheString() const;
    void settheString(const IT_Bus::String& val);

    enum PolyDiscriminator
    {
        theShort,
        theString,
        Poly_MAXLONG=-1L
    } m_discriminator;

    PolyDiscriminator get_discriminator() const { ... }
    IT_Bus::UInt get_discriminator_as_uint() const { ... }
    ...
};
```

The value of the union can be modified and accessed using the `get`*`UnionMember`*`()` and `set`*`UnionMember`*`()` pairs of functions. The union discriminator can be accessed through the `get_discriminator()` and `get_discriminator_as_uint()` functions.

---

struct type

Consider the following definition of an IDL struct type, `SampleTypes::SampleStruct`:

```
// IDL
module SampleTypes {
    struct SampleStruct {
        string theString;
        long theLong;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStruct` struct to an XML schema sequence complex type, `SampleTypes.SampleStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SampleStruct">
    <xsd:sequence>
        <xsd:element name="theString" type="xsd:string"/>
        <xsd:element name="theLong" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-C++ compiler maps the `SampleTypes.SampleStruct` type to a C++ class, `SampleTypes_SampleStruct`, as follows:

```
class SampleTypes_SampleStruct : public
   IT_Bus::SequenceComplexType
{
  public:
    SampleTypes_SampleStruct();
    SampleTypes_SampleStruct(const SampleTypes_SampleStruct&
   copy);
    ...
    const IT_Bus::String & gettheString() const;
    IT_Bus::String & gettheString();
    void settheString(const IT_Bus::String & val);

    const IT_Bus::Int & gettheLong() const;
    IT_Bus::Int & gettheLong();
    void settheLong(const IT_Bus::Int & val);
};
```

The members of the struct can be accessed and modified using the `get`*`StructMember`*`()` and `set`*`StructMember`*`()` pairs of functions.

---

sequence types

Consider the following definition of an IDL sequence type, `SampleTypes::SeqOfStruct`:

```
// IDL
module SampleTypes {
    typedef sequence< SampleStruct > SeqOfStruct;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SeqOfStruct` sequence to a WSDL sequence type with occurrence constraints, `SampleTypes.SeqOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SeqOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-C++ compiler maps the `SampleTypes.SeqOfStruct` type to a C++ class, `SampleTypes_SeqOfStruct`, as follows:

```
class SampleTypes_SeqOfStruct : public
   IT_Bus::ArrayT<SampleTypes_SampleStruct,
   &SampleTypes_SeqOfStruct_item_qname, 0, -1>
{
  public:
    ...
};
```

The `SampleTypes_SeqOfStruct` class is an Artix C++ array type (based on the `IT_Vector` template). Hence, the array class has an API similar to the `std::vector` type from the C++ Standard Template Library.

Note: IDL bounded sequences map in a similar way to normal IDL sequences, except that the `IT_Bus::ArrayT` base class uses the bounds specified in the IDL.

---

array types

Consider the following definition of an IDL union type, `SampleTypes::ArrOfStruct`:

```
// IDL
module SampleTypes {
     typedef SampleStruct ArrOfStruct[10];
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::ArrOfStruct` array to a WSDL sequence type with occurrence constraints, `SampleTypes.ArrOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.ArrOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="10" maxOccurs="10"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-C++ compiler maps the `SampleTypes.ArrOfStruct` type to a C++ class, `SampleTypes_ArrOfStruct`, as follows:

```
class SampleTypes_ArrOfStruct : public
   IT_Bus::ArrayT<SampleTypes_SampleStruct,
   &SampleTypes_ArrOfStruct_item_qname, 10, 10>
{
    ...
};
```

The `SampleTypes_ArrOfStruct` class is an Artix C++ array type (based on the `IT_Vector` template). The array class has an API similar to the `std::vector` type from the C++ Standard Template Library, except that the size of the vector is restricted to the specified array length, `10`.

---

exception types

Consider the following definition of an IDL exception type, `SampleTypes::GenericException`:

```
// IDL
module SampleTypes {
    exception GenericExc {
        string reason;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::GenericExc` exception to a WSDL sequence type, `SampleTypes.GenericExc`, and to a WSDL fault message, `_exception.SampleTypes.GenericExc`, as follows:

```
<xsd:complexType name="SampleTypes.GenericExc">
    <xsd:sequence>
        <xsd:element name="reason" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
...
<xsd:element name="SampleTypes.GenericExc"
    type="xsd1:SampleTypes.GenericExc"/>
...
<message name="_exception.SampleTypes.GenericExc">
    <part name="exception"
   element="xsd1:SampleTypes.GenericExc"/>
</message>
```

The WSDL-to-C++ compiler maps the `SampleTypes.GenericExc` and `_exception.SampleTypes.GenericExc` types to C++ classes, `SampleTypes_GenericExc` and `_exception_SampleTypes_GenericExc`, as follows:

```
// C++
class SampleTypes_GenericExc : public
   IT_Bus::SequenceComplexType
{
  public:
    SampleTypes_GenericExc();
    ...
    const IT_Bus::String & getreason() const;
    IT_Bus::String & getreason();
    void setreason(const IT_Bus::String & val);
};
...
class _exception_SampleTypes_GenericExcException : public
   IT_Bus::UserFaultException
{
  public:
    _exception_SampleTypes_GenericExcException();
    ...
    const SampleTypes_GenericExc & getexception() const;
    SampleTypes_GenericExc & getexception();
    void setexception(const SampleTypes_GenericExc & val);
    ...
};
```

typedef of a simple type

Consider the following IDL typedef that defines an alias of a `float`, `SampleTypes::FloatAlias`:

```
// IDL
module SampleTypes {
    typedef float FloatAlias;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::FloatAlias` typedef directory to the type, `xsd:float`.

The WSDL-to-C++ compiler then maps the `xsd:float` type directly to the `IT_Bus::Float` C++ type. Hence, no C++ typedef is generated for the `float` type.

typedef of a complex type

Consider the following IDL typedef that defines an alias of a `struct`, `SampleTypes::SampleStructAlias`:

```
// IDL
module SampleTypes {
    typedef SampleStruct SampleStructAlias;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStructAlias` typedef directly to the plain, unaliased `SampleTypes.SampleStruct` type.

The WSDL-to-C++ compiler then maps the `SampleTypes.SampleStruct` WSDL type directly to the `SampleTypes::SampleStruct` C++ type. Hence, no C++ typedef is generated for this struct type. Instead of a typedef, the C++ mapping uses the original, unaliased type.

Note: The typedef of an IDL sequence or an IDL array is treated as a special case, with a specific C++ class being generated to represent the sequence or array type.

IDL Module and Interface Mapping

Overview

This section describes the Artix C++ mapping for the following IDL constructs:

- Module mapping.
- Interface mapping.
- Object reference mapping.
- Operation mapping.
- Attribute mapping.

Module mapping

An IDL identifier appearing within the scope of an IDL module, *ModuleName*::*Identifier*, maps to a C++ identifier of the form *ModuleName_Identifier*. That is, the IDL scoping operator, ::, maps to an underscore, _, in C++.

Although IDL modules do *not* map to namespaces under the Artix C++ mapping, it is possible nevertheless to put generated C++ code into a namespace using the `-n` switch to the WSDL-to-C++ compiler.

For example, if you pass a namespace, `TEST`, to the WSDL-to-C++ `-n` switch, the *ModuleName*::*Identifier* IDL identifier would map to `TEST`::*ModuleName_Identifier*.

Interface mapping

An IDL interface, *InterfaceName*, maps to a C++ class of the same name, *InterfaceName*. If the interface is defined in the scope of a module, that is *ModuleName*::*InterfaceName*, the interface maps to the *ModuleName_InterfaceName* C++ class.

If an IDL data type, *TypeName*, is defined within the scope of an IDL interface, that is *ModuleName*::*InterfaceName*::*TypeName*, the type maps to the *ModuleName_InterfaceName_TypeName* C++ class.

Object reference mapping

When an IDL interface is used as an operation parameter or return type, it is mapped to the `IT_Bus::Reference` C++ type.

For example, consider an operation, `get_foo()`, that returns a reference to a `Foo` interface as follows:

```
// IDL
interface Foo {};

interface Bar {
    Foo get_foo();
};
```

The `get_foo()` IDL operation then maps to the following C++ function:

```
// C++
void get_foo(
    IT_Bus::Reference & var_return
) IT_THROW_DECL((IT_Bus::Exception));
```

Note that this mapping is very different from the OMG IDL-to-C++ mapping. In the Artix mapping, the `get_foo()` operation does not return a pointer to a `Foo` proxy object. Instead, you must construct the `Foo` proxy object in a separate step, by passing the `IT_Bus::Reference` object into the `FooClient` constructor.

Operation mapping

Example 10 shows two IDL operations defined within the `SampleTypes::Foo` interface. The first operation is a regular IDL operation, `test_op()`, and the second operation is a oneway operation, `test_oneway()`.

Example 1: *Example IDL Operations*

```
// IDL
module SampleTypes {
    ...
    interface Foo {
        ...
        SampleStruct test_op(
            in SampleStruct    in_struct,
            inout SampleStruct inout_struct,
            out SampleStruct   out_struct
        ) raises (GenericExc);

        oneway void test_oneway(in string in_str);
    };
};
```

The operations from the preceding IDL, Example 10 on page 98, map to C++ as shown in Example 11,

Example 2: *Mapping IDL Operations to C++*

```
// C++
class SampleTypes_Foo
{
  public:
    ...
1   virtual void test_op(
        const TEST::SampleTypes_SampleStruct & in_struct,
        TEST::SampleTypes_SampleStruct & inout_struct,
        TEST::SampleTypes_SampleStruct & var_return,
        TEST::SampleTypes_SampleStruct & out_struct
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;

2   virtual void test_oneway(
        const IT_Bus::String & in_str
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;
};
```

The preceding C++ operation signatures can be explained as follows:

1. The C++ mapping of an IDL operation always has the return type `void`. If a return value is defined in IDL, it is mapped as an out parameter, `var_return`.

   The order of parameters in the C++ function signature, `test_op()`, is determined as follows:

   - First, the in and inout parameters appear in the same order as in IDL, ignoring the out parameters.
   - Next, the return value appears as the parameter, `var_return` (with the same semantics as an out parameter).
   - Finally, the out parameters appear in the same order as in IDL, ignoring the in and inout parameters.

2. The C++ mapping of an IDL oneway operation is straightforward, because a oneway operation can have only `in` parameters and a `void` return type.

Attribute mapping

Example 12 shows two IDL attributes defined within the `SampleTypes::Foo` interface. The first attribute is readable and writable, `str_attr`, and the second attribute is readonly, `struct_attr`.

Example 3: *Example IDL Attributes*

```
// IDL
module SampleTypes {
    ...
    interface Foo {
        ...
        attribute string               str_attr;
        readonly attribute SampleStruct struct_attr;
    };
};
```

The attributes from the preceding IDL, Example 12 on page 99, map to C++ as shown in Example 13,

Example 4: *Mapping IDL Attributes to C++*

```
// C++
class SampleTypes_Foo
{
  public:
    ...
1   virtual void _get_str_attr(
        IT_Bus::String & var_return
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;

    virtual void _set_str_attr(
        const IT_Bus::String & _arg
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;

2   virtual void _get_struct_attr(
        TEST::SampleTypes_SampleStruct & var_return
    ) IT_THROW_DECL((IT_Bus::Exception)) = 0;
};
```

The preceding C++ attribute signatures can be explained as follows:

1. A normal IDL attribute, *`AttributeName`*, maps to a pair of accessor and modifier functions in C++, `_get_`*`AttributeName`*`()`, `_set_`*`AttributeName`*`()`.
2. An IDL readonly attribute, *`AttributeName`*, maps to a single accessor function in C++, `_get_`*`AttributeName`*`()`.

CHAPTER 7

Artix WSDL-to-IDL Mapping

This chapter describes how the Artix WSDL-to-IDL compiler maps WSDL types to OMG IDL types.

In this chapter

This chapter discusses the following topics:

Simple Types

Overview

This section describes the mapping of simple WSDL types to IDL.

In this section

This section contains the following subsections:

Atomic Types

Table of atomic types

Table 2 shows how the XSD schema atomic types map to IDL.

Table 1: *XSD Schema Simple Types Mapping to IDL*

XSD Schema Type	IDL Type
xsd:boolean	boolean
xsd:byte	char
xsd:unsignedByte	octet
xsd:short	short
xsd:unsignedShort	unsigned short
xsd:int	long
xsd:unsignedInt	unsigned long
xsd:long	long long
xsd:unsignedLong	unsigned long long
xsd:float	float
xsd:double	double
xsd:string	string
xsd:normalizedString	string
xsd:token	string
xsd:language	string
xsd:NMTOKEN	string
xsd:NMTOKENS	*Not supported*
xsd:Name	string
xsd:NCName	string

Table 1: *XSD Schema Simple Types Mapping to IDL*

XSD Schema Type	IDL Type
`xsd:ID`	`string`
`xsd:QName`	`string`
`xsd:dateTime`	`TimeBase::UtcT`
`xsd:date`	`string`
`xsd:time`	`string`
`xsd:gDay`	`string`
`xsd:gMonth`	`string`
`xsd:gMonthDay`	`string`
`xsd:gYear`	`string`
`xsd:gYearMonth`	`string`
`xsd:decimal`	*Typedef of* `fixed<31,6>`
`xsd:integer`	`long long`
`xsd:positiveInteger`	`unsigned long long`
`xsd:negativeInteger`	`long long`
`xsd:nonPositiveInteger`	`long long`
`xsd:nonNegativeInteger`	`unsigned long long`
`xsd:base64Binary`	`base64BinarySeq` *(typedef of* `sequence<octet>`*)*
`xsd:hexBinary`	`hexBinarySeq` *(typedef of* `sequence<octet>`*)*
`soapenc:base64`	`base64Seq` *(typedef of* `sequence<octet>`*)*
`xsd:ID`	*Not supported.*

String Type

Overview

By default, `xsd:string` maps to the ordinary IDL `string` type.

If you are planning to use international strings, however, you might want `xsd:string` to map to the IDL wide string type, `wstring`, instead. The `wsdltocorba` utility does not provide an option to change the default mapping, but you can easily alter the mapping by manually editing the contents of the CORBA `<binding>` tag in the WSDL.

Default CORBA binding

Consider, for example, how to add a CORBA binding to the `Greeter` port type (see the `hello_world.wsdl` file located in *ArtixInstallDir*`/artix/`*Version*`/demos/basic/hello_world_soap_http/etc`). You can add a CORBA binding by entering the following command:

```
> wsdltocorba -corba -i Greeter hello_world.wsdl
```

The WSDL output from this command, `hello_world-corba.wsdl`, includes a new CORBA binding, `GreeterCORBABinding`, as shown in Example 14. The contents of this `binding` element essentially determine the WSDL-to-CORBA mapping for the port type. Some parameters and return types in the binding are declared to have an `idltype` attribute of `corba:string`, which means they map to the IDL `string` type.

Example 1: *Default CORBA Binding Generated by wsdltocorba*

```
<definitions ... >
    ...
    <binding name="GreeterCORBABinding" type="tns:Greeter">
        <corba:binding repositoryID="IDL:Greeter:1.0"/>
        <operation name="sayHi">
            <corba:operation name="sayHi">
                <corba:return idltype="corba:string" name="theResponse"/>
            </corba:operation>
            <input name="sayHiRequest"/>
            <output name="sayHiResponse"/>
        </operation>
        <operation name="greetMe">
            <corba:operation name="greetMe">
                <corba:param idltype="corba:string" mode="in" name="me"/>
                <corba:return idltype="corba:string" name="theResponse"/>
```

Example 1: *Default CORBA Binding Generated by wsdltocorba*

```
            </corba:operation>
            <input name="greetMeRequest"/>
            <output name="greetMeResponse"/>
        </operation>
    </binding>
</definitions>
```

Manually modified CORBA binding

To alter the WSDL-to-IDL string mapping, replace some or all of the instances of `corba:string` by `corba:wstring`. Example 15 shows the result of replacing all instances of `corba:string` by `corba:wstring`.

Example 2: *Manually Modified CORBA Binding*

```
<definitions ... >
    ...
    <binding name="GreeterCORBABinding" type="tns:Greeter">
        <corba:binding repositoryID="IDL:Greeter:1.0"/>
        <operation name="sayHi">
            <corba:operation name="sayHi">
                <corba:return idltype="corba:wstring" name="theResponse"/>
            </corba:operation>
            <input name="sayHiRequest"/>
            <output name="sayHiResponse"/>
        </operation>
        <operation name="greetMe">
            <corba:operation name="greetMe">
                <corba:param idltype="corba:wstring" mode="in" name="me"/>
                <corba:return idltype="corba:wstring" name="theResponse"/>
            </corba:operation>
            <input name="greetMeRequest"/>
            <output name="greetMeResponse"/>
        </operation>
    </binding>
</definitions>
```

Generated IDL

Example 16 shows the IDL that would be generated from the modified CORBA binding in Example 15 on page 106.

Example 3: *IDL Generated from the Modified CORBA Binding*

```
// IDL

interface Greeter {
    wstring sayHi();
    wstring greetMe(in wstring me);
};
```

To generate this IDL interface, you would enter the following command:

```
> wsdltocorba -idl -b GreeterCORBABinding hello_world-corba.wsdl
```

Date and Time Types

Overview

The WSDL-to-IDL mapping currently supports only the `xsd:dateTime` type, which maps to the `TimeBase::UtcT` IDL type.

Note: The mapping is subject to certain restrictions, as detailed below.

TimeBase::UtcT type

The `TimeBase::UtcT` type, which holds a UTC time value, is defined in the OMG's *CORBA Time Service* specification. Example 17 shows the definition of `UtcT` in the `TimeBase` module.

Example 4: *Definition of the TimeBase IDL Module*

```
// IDL
module TimeBase
{
    typedef unsigned long long TimeT;
    typedef TimeT              InaccuracyT;
    typedef short              TdfT;

    struct UtcT
    {
        TimeT          time;
        unsigned long  inacclo;
        unsigned short inacchi;
        TdfT           tdf;
    };

    struct IntervalT
    {
        TimeT lower_bound;
        TimeT upper_bound;
    };
};
```

Unsupported time/date values

The following `xsd:dateTime` values cannot be mapped to `TimeBase::UtcT`:

- Values with a local time zone. Local time is treated as a 0 UTC time zone offset.
- Values prior to 15 October 1582.
- Values greater than approximately 30,000 A.D.

The following `TimeBase::UtcT` values cannot be mapped to `xsd:dateTime`:

- Values with a non-zero `inacclo` or `inacchi`.
- Values with a time zone offset that is not divisible by 30 minutes.
- Values with time zone offsets greater than 14:30 or less than -14:30.
- Values with greater than millisecond accuracy.
- Values with years greater than 9999.

Deriving Simple Types by Restriction

Overview

Most derived simple types are mapped as if they had been declared to be the base type. For example, XSD types derived from `xsd:string` are treated as if they were declared as `xsd:string` and are therefore mapped to the IDL `string` type.

Exceptionally, derived simple types declared using the `<enumeration>` facet are treated as a special case: enumerated simple types are mapped to an IDL `enum` type.

Unchecked facets

The following facets can be used, but are not checked at runtime:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `enumeration`
- `whiteSpace`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`
- `totalDigits`
- `fractionDigits`

Checked facets

The following facets are supported and checked at runtime:

- `enumeration`

Example with a maxLength facet

The following example shows how you can use the <maxLength> facet to define a string whose length is limited to 100 characters:

```
<xsd:simpleType name="String100">
    <xsd:restriction base="xsd:string">
        <xsd:maxLength value="100"/>
    </xsd:restriction>
</xsd:simpleType>
```

The WSDL-to-IDL mapping maps this String100 type to the string type.

Example with enumeration facets

The following example shows how to define an enumerated type, ColorEnum, using the <enumeration> facet:

```
<xsd:simpleType name="ColorEnum">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="RED"/>
        <xsd:enumeration value="GREEN"/>
        <xsd:enumeration value="BLUE"/>
    </xsd:restriction>
</xsd:simpleType>
```

The WSDL-to-IDL mapping maps this ColorEnum type to the following IDL enum type.

```
// IDL
enum ColorEnum {
    RED,
    GREEN,
    BLUE
};
```

List Type

Overview

An `xsd:list` type maps to an IDL sequence type, `sequence<`*`MappedElementType`*`>`, where *`MappedElementType`* is the IDL type representing the list elements.

There are two styles of list declaration, both of which are supported in Artix:

- Lists defined using itemType.
- Lists defined by derivation.

Lists defined using itemType

Where the list element type is a schema atomic type, you can define the list type using the `itemType` attribute. For example, a list of strings can be defined as follows:

```
<xsd:simpleType name="StringList">
    <xsd:list itemType="xsd:string"/>
</xsd:simpleType>
```

This maps to the following IDL type:

```
// IDL
typedef sequence<string> StringList;
```

Lists defined by derivation

Where the list element type is derived from a schema atomic type (by the application of various restricting facets), you can define the list type using a `restriction` element. For example, you can define a list of restricted integers as follows:

```
<xsd:simpleType name="IntList">
    <xsd:list>
        <xsd:simpleType>
            <xsd:restriction base="xsd:int">
                <xsd:maxInclusive value="1000"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:list>
</xsd:simpleType>
```

This maps to the following IDL type:

```
// IDL
typedef sequence<long> IntList;
```

Unsupported Simple Types

Overview

This subsection lists the XSD simple types that are not supported by the `wsdltocorba` mapping utility.

Unsupported types

The following XSD simple types are not supported by the WSDL-to-IDL mapping:

```
xsd:duration
xsd:ENTITY
xsd:ENTITIES
xsd:IDREF
xsd:IDREFS
xsd:NMTOKENS
xsd:NOTATION
xsd:union
```

Complex Types

Overview

This section describes the mapping of complex WSDL types to IDL.

In this section

This section contains the following subsections:

Sequence Complex Types

Overview

The XSD sequence complex type maps to an IDL `struct` type, where each element of the original sequence maps to a member of the IDL `struct`.

Occurrence constraints

The WSDL-to-IDL mapping does *not* support occurrence constraints on the `sequence` element. If `minOccurs` or `maxOccurs` attribute settings appear in the `sequence` element, they are ignored by the WSDL-to-IDL compiler.

On the other hand, elements appearing *within* the `sequence` element can define occurrence constraints—see "Arrays" on page 128.

WSDL example

Example 18 shows an XSD sequence type with three simple elements.

Example 5: *Definition of a Sequence Complex Type in WSDL*

```
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

IDL mapping

Example 19 shows the result of mapping the `SimpleStruct` type (from the preceding Example 18) to IDL.

Example 6: *Mapping of SimpleStruct to IDL*

```
// IDL
struct SimpleStruct {
    float varFloat;
    long varInt;
    string varString;
};
```

Choice Complex Types

Overview

The XSD choice complex type maps to an IDL `union` type, where each element of the original choice maps to a member of the IDL `union`.

Occurrence constraints

Artix does not support occurrence constraints on the `choice` element.

WSDL example

Example 20 shows an XSD choice type with three elements.

Example 7: *Definition of a Choice Complex Type in WSDL*

```
<xsd:complexType name="SimpleChoice">
    <xsd:choice>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

IDL mapping

Example 21 shows the result of mapping the `SimpleChoice` type (from the preceding Example 20) to IDL.

Example 8: *Mapping of SimpleChoice to IDL*

```
// IDL
union SimpleChoice switch (long) {
    case 0:
        float varFloat;
    case 1:
        long varInt;
    case 2:
        string varString;
};
```

All Complex Types

Overview

The XSD all complex type maps to an IDL `struct` type, where each element of the original all maps to a member of the IDL `struct`.

Occurrence constraints

Artix does not support occurrence constraints on the `all` element.

WSDL example

Example 22 shows an XSD all type with three simple elements.

Example 9: *Definition of an All Complex Type in WSDL*

```
<xsd:complexType name="SimpleAll">
    <xsd:all>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:all>
</xsd:complexType>
```

IDL mapping

Example 23 shows the result of mapping the `SimpleAll` type (from the preceding Example 22) to IDL.

Example 10: *Mapping of SimpleAll to IDL*

```
// IDL
struct SimpleAll {
    float varFloat;
    long varInt;
    string varString;
};
```

Attributes

Overview

Attributes of a sequence type or of an all type map to additional members of an IDL `struct`. The type representing an attribute in IDL is defined as a *nillable type* (see "Nillable Types" on page 134 for details). This makes it possible for attributes to be treated as optional.

Attributes can be declared within the scope of the `xsd:complexType` element. Hence, you can include attributes in the definitions of an all type, a sequence type, and a choice type.

Note: Attributes of a choice type are currently *not* supported by the WSDL-to-IDL mapping.

The declaration of an attribute in a complex type has the following syntax:

```
<xsd:complexType name="TypeName">
    <xsd:attribute name="AttrName" type="AttrType"
                   use="[optional|required|prohibited]"/>
    ...
</xsd:complexType>
```

Attribute use

The `use` attribute setting is ignored by the WSDL-to-IDL mapping.

Because attributes are declared as nillable types in IDL, the attributes are effectively `optional` by default. If the attribute `use` is defined as `required` or `prohibited`, however, it is up to the developer to enforce these conditions.

WSDL example

Example 24 shows an XSD sequence type, which is declared to have two attributes, `varAttrString` and `varAttrIntOptional`.

Example 11: *Definition of a Complex Type with Attributes in WSDL*

```
<xsd:complexType name="SimpleStructWithAttributes">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="varAttrString" type="xsd:string"/>
    <xsd:attribute name="varAttrIntOptional" type="xsd:int"
                   use="optional"/>
</xsd:complexType>
```

IDL mapping

Example 25 shows the result of mapping the `SimpleStructWithAttributes` type (from the preceding Example 24) to IDL.

Example 12: *Mapping of SimpleStructWithAttributes to IDL*

```
// IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
union long_nil switch(boolean) {
    case TRUE:
        long value;
};

struct SimpleStructWithAttributes {
    string_nil varAttrString;
    long_nil varAttrIntOptional;
    float varFloat;
    long varInt;
    string varString;
};
```

Nesting Complex Types

Overview

It is possible to nest complex types within each other. When mapped to IDL, the nested complex types map to a nested hierarchy of structs, where each instance of a nested type is declared as a member of another struct.

Avoiding anonymous types

In general, it is recommended that you name types that are nested inside other types, instead of using anonymous types. This results in simpler code when the types are mapped to IDL.

Note: The WSDL-to-IDL mapping has only limited supported for mapping anonymous type, which does not work in all cases.

WSDL example

Example 26 shows the definition of a nested sequence type, `NestedStruct`, which contains another sequence type, `SimpleStruct`, as an element.

Example 13: *Definition of a Nested Type in WSDL*

```
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="NestedStruct">
    <xsd:sequence>
        <xsd:element name="varString" type="xsd:string"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varStruct" type="tns:SimpleStruct"/>
    </xsd:sequence>
</xsd:complexType>
```

IDL mapping

Example 27 shows the result of mapping the `NestedStruct` type (from the preceding Example 26) to IDL.

Example 14: *Mapping of NestedStruct to IDL*

```
// IDL
struct SimpleStruct {
    float varFloat;
    long varInt;
    string varString;
};

struct NestedStruct {
    string varString;
    long varInt;
    float varFloat;
    SimpleStruct varStruct;
};
```

Deriving a Complex Type from a Simple Type

Overview

A complex type derived from a simple type maps to an IDL `struct` type with a member, `_simpleTypeValue`, to hold the value of the simple type. Any attributes defined by the derived type are represented as nillable members of the struct (see "Attributes" on page 119 for more details).

The following kinds of derivation are supported:

- Derivation by restriction.
- Derivation by extension.

Derivation by restriction

Example 28 shows an example of a complex type, `OrderNumber`, derived by restriction from the `xsd:decimal` simple type. The new type is restricted to have values less than 1,000,000.

Example 15: *Complex Type Derived by Restriction from a Simple Type*

```
<xsd:complexType name="OrderNumber">
    <xsd:simpleContent>
        <xsd:restriction base="xsd:decimal">
            <xsd:maxExclusive value="1000000"/>
        </xsd:restriction>
    </xsd:simpleContent>
</xsd:complexType>
```

IDL mapping of restricted type

Example 29 shows the result of mapping the `OrderNumber` type (from the preceding Example 28) to IDL. The `_simpleTypeValue` struct member represents the simple type value.

Example 16: *Mapping of OrderNumber to IDL*

```
// IDL
typedef fixed<31, 6> fixed_1;

struct OrderNumber {
    fixed_1 _simpleTypeValue;
};
```

Derivation by extension

Example 30 shows an example of a complex type, `InternationalPrice`, derived by extension from the `xsd:decimal` simple type. The new type is extended to include a currency attribute.

Example 17: *Complex Type Derived by Extension from a Simple Type*

```
<xsd:complexType name="InternationalPrice">
    <xsd:simpleContent>
        <xsd:extension base="xsd:decimal">
            <xsd:attribute name="currency" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

IDL mapping of extended type

Example 31 shows the result of mapping the `InternationalPrice` type (from the preceding Example 30) to IDL. In addition to the `_simpleTypeValue` member, representing the simple type, there is a `currency` member of `string_nil` type, representing the currency attribute.

Example 18: *Mapping of InternationalPrice to IDL*

```
// IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
typedef fixed<31, 6> fixed_1;

struct InternationalPrice {
    string_nil currency;
    fixed_1 _simpleTypeValue;
};
```

Deriving a Complex Type from a Complex Type

Overview

Artix supports derivation of a complex type from a complex type, for which the following kinds of derivation are possible:

- Derivation by restriction.
- Derivation by extension.

Allowed inheritance relationships

Figure 16 shows the inheritance relationships allowed between complex types. All of these inheritance relationships are supported by the WSDL-to-IDL mapping, including cross-inheritance. For example, a sequence can derive from a choice, a choice from an all, an all from a choice, and so on.

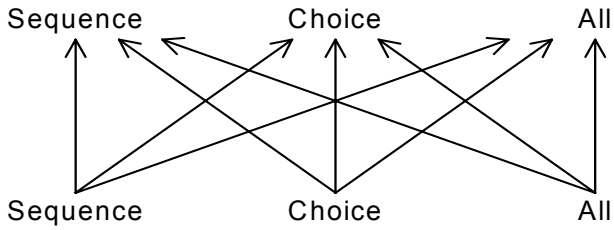

Figure 1: *Allowed Inheritance Relationships for Complex Types*

IDL mapping

Artix maps schema derived types to an IDL struct (irrespective of whether the schema derived type is a `sequence`, a `choice`, or an `all`). The generated IDL struct always contains the following two members:

- *The base member*—holds an instance of the base type, *`BaseType`*. The name of this member is *`BaseType`*`_f`.
- *The extension member*—holds an instance of the extension type. The name of this member obeys the following naming convention (where *`DerivedType`* is the name of the derived type in XML):
  - `sequence` extension—the name is *`DerivedType`*`SequenceStruct_f`.
  - `choice` extension—the name is *`DerivedType`*`ChoiceType_f`.
  - `all` extension—the name is *`DerivedType`*`AllStruct_f`.

In addition, if the derived type defines attributes, they are mapped directly to members of the IDL struct.

WSDL example

Example 32 shows the definition of a derived type that is obtained by extending a `sequence` type (base type) with a `choice` type (extension type).

Example 19: *XML Example of a Choice Type Derived from a Struct Type*

```
// Base type.
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

// Derived type.
<xsd:complexType name="DerivedChoice_BaseStruct">
    <xsd:complexContent mixed="false">
        <xsd:extension base="s:SimpleStruct">
            <xsd:choice>
                <xsd:element name="varStringExt"
                             type="xsd:string"/>
               <xsd:element name="varFloatExt" type="xsd:float"/>
            </xsd:choice>
            <xsd:attribute name="attrString" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
```

Example 19: *XML Example of a Choice Type Derived from a Struct Type*

```
</xsd:complexType>
```

Mapped example

The preceding `DerivedChoice_BaseStruct` schema type maps to an IDL struct, `DerivedChoice_BaseStruct`, as shown in Example 33.

Example 20: *IDL Mapping of the DerivedChoice_BaseStruct Type*

```
// IDL

// Base type.
struct SimpleStruct {
    float varFloat;
    long varInt;
    string varString;
};

// Extended part of derived type.
union DerivedChoice_BaseStructChoiceType switch(long) {
    case 0:
        string varStringExt;
    case 1:
        float varFloatExt;
};

// Derived type.
struct DerivedChoice_BaseStruct {
    string_nil attrString;

    SimpleStruct SimpleStruct_f;
    DerivedChoice_BaseStructChoiceType
        DerivedChoice_BaseStructChoiceType_f;
};
```

Arrays

Overview

An Artix array is a sequence complex type that satisfies the following special conditions:

- The sequence complex type schema defines a *single* element only.
- The element definition has a `maxOccurs` attribute with a value greater than 1.

Note: All elements implicitly have `minOccurs=1` and `maxOccurs=1`, unless specified otherwise.

Hence, an Artix array definition has the following general syntax:

```
<complexType name="ArrayName">
    <sequence>
        <element name="ElemName" type="ElemType"
                 minOccurs="LowerBound" maxOccurs="UpperBound"/>
    </sequence>
</complexType>
```

The *ElemType* specifies the type of the array elements and the number of elements in the array can be anywhere in the range *LowerBound* to *UpperBound*.

Mapping arrays to IDL

The way Artix maps arrays to IDL depend on the values of the `minOccurs` and `maxOccurs` attributes, as shown in Table 3.

Table 2: *Array to IDL Mapping for Various Occurrence Constraints*

Occurrence Constraints	IDL Type
`minOccurs="N" maxOccurs="N"`	*ArrayName*[*N*]
`minOccurs="N" maxOccurs="M"` (with *N* < *M*)	sequence<*ElementType*, *M*>
`maxOccurs="unbounded"`	sequence<*ElementType*>

Fixed array

The following XSD schema shows the definition of an array, `FixedArray`, whose `minOccurs` and `maxOccurs` constraints are set to an identical, finite value.

```
<xsd:complexType name="FixedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="3" minOccurs="3"
                     name="item" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `FixedArray` schema type maps to the following IDL array:

```
// IDL
typedef long FixedArray[3];
```

Bounded array

The following XSD schema shows the definition of an array, `BoundedArray`, whose `minOccurs` and `maxOccurs` constraints are finite and unequal.

```
<xsd:complexType name="BoundedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="3" minOccurs="1"
                     name="item" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `BoundedArray` schema type maps to the following IDL bounded sequence type:

```
// IDL
typedef sequence<float, 3> BoundedArray;
```

Unbounded array

The following XSD schema shows the definition of an array, `UnboundedArray`, whose `maxOccurs` constraint is unbounded.

```
<xsd:complexType name="UnboundedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="item" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `UnboundedArray` schema type maps to the following IDL unbounded sequence type:

```
// IDL
typedef sequence<string> UnboundedArray;
```

Nested arrays

The following XSD schema shows the definition of a nested array, `NestedArray`, which is defined as an array whose elements are of `UnboundedArray` type.

```
<xsd:complexType name="NestedArray">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="subarray" type="s:UnboundedArray"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `NestedArray` schema type maps to the following IDL unbounded sequence type:

```
// IDL
typedef sequence<UnboundedArray> NestedArray;
```

Wildcarding Types

Overview

The XML schema wildcarding types enable you to define XML types with loosely defined characteristics. Table 4 shows how the XSD schema wildcarding types map to IDL.

Table 3: *XSD Schema Simple Types Mapping to IDL*

XSD Schema Type	IDL Type
`xsd:anyURI`	`string`
`xsd:anyType`	`any`
`xsd:any`	*Not supported*

xsd:anyType example

Consider an XSD sequence, `AnyStruct`, whose elements are declared to be of `xsd:anyType` type, as shown in Example 34.

Example 21: *AnyStruct Schema Type with xsd:anyType Members*

```
<xsd:complexType name="AnyStruct">
    <xsd:sequence>
        <xsd:element name="varAny_1" type="xsd:anyType"/>
        <xsd:element name="varAny_2" type="xsd:anyType"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `AnyStruct` schema type maps to the IDL struct type shown in Example 35.

Example 22: *Mapping of AnyStruct Type to IDL*

```
struct AnyStruct {
    any varAny_1;
    any varAny_2;
};
```

Occurrence Constraints

Overview

Certain XML schema tags—for example, `<element>`, `<sequence>`, `<choice>` and `<any>`—can be declared to occur multiple times using *occurrence constraints*. The occurrence constraints are specified by assigning integer values (or the special value `unbounded`) to the `minOccurs` and `maxOccurs` attributes.

The WSDL-to-IDL mapping currently supports only *element occurrence constraints* (that is, `minOccurs` and `maxOccurs` attribute settings within the `<element>` tag).

Element occurrence constraints

You define occurrence constraints on a schema element by setting the `minOccurs` and `maxOccurs` attributes for the element. Hence, the definition of an element with occurrence constraints in an XML schema element has the following form:

```
<element name="ElemName" type="ElemType" minOccurs="LowerBound"
   maxOccurs="UpperBound"/>
```

Note: When a sequence schema contains a *single* element definition and this element defines occurrence constraints, it is treated as an array. See "Arrays" on page 128.

Limitations

In the current version of Artix, element occurrence constraints can be used only within the following complex types:

- `all` complex types,
- `sequence` complex types.

Element occurrence constraints are *not* supported within the scope of the following:

- `choice` complex types.

Mapping to IDL

Given an `<xsd:element name="ElemName" ... >` element with occurrence constraints, defined in an `<xsd:sequence>` or an `<xsd:all>` tag, Artix defines an *ElemName*`Array` type in IDL to represent the multiply occurring element.

The *ElemName*Array type is defined according to the rules in Table 3 on page 128, which determine the mapped IDL type based on the values of the minOccurs and maxOccurs attributes.

Example of element occurrence constraints

The following XSD schema shows the definition of an <xsd:sequence> type, CompoundArray, which has two multiply occurring member elements.

```
<xsd:complexType name="CompoundArray">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="array1" type="xsd:string"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
                     name="array2" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding CompoundArray schema type maps to the following IDL struct, CompoundArray, which uses two generated array types, array1Array and array2Array, to represent the types of its member elements:

```
// IDL
typedef sequence<string> array1Array;
typedef sequence<string> array2Array;

struct CompoundArray {
    array1Array array1;
    array2Array array2;
};
```

Nillable Types

Overview

An element in an XML schema may be declared as nillable by setting the `nillable` attribute equal to `true`. This is useful in cases where you would like to have the option of transmitting no value for a type (for example, if you would like to define an operation with optional parameters).

Nillable syntax

To declare an element as nillable, use the following syntax:

```
<element name="ElementName" type="ElementType" nillable="true"/>
```

The `nillable="true"` setting indicates that this as a nillable element. If the `nillable` attribute is missing, the default is value is `false`.

Mapping to IDL

If a given element of *`ElementType`* type is defined with `nillable="true"` and *`ElementType`* maps to *`MappedType`* in IDL, Artix automatically generates a union IDL type, *`MappedType`*`_nil`, as follows:

```
// IDL
union MappedType_nil switch(boolean) {
    case TRUE:
        MappedType value;
};
```

Artix uses this *`MappedType`*`_nil` type to represent the type of the nillable element in IDL (for example, where it appears as the member of a struct and so on).

Example

The following XSD schema shows the definition of an `<xsd:sequence>` type, `StructWithNillables`, which contains several nillable elements:

```
<xsd:complexType name="SimpleStruct">
    <xsd:sequence>
        <xsd:element name="varFloat" type="xsd:float"/>
        <xsd:element name="varInt" type="xsd:int"/>
        <xsd:element name="varString" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="varAttrString" type="xsd:string"/>
</xsd:complexType>
```

```
<xsd:complexType name="StructWithNillables">
    <xsd:sequence>
        <xsd:element name="varFloat" nillable="true"
                     type="xsd:float"/>
        <xsd:element name="varInt" nillable="true"
                     type="xsd:int"/>
        <xsd:element name="varString" nillable="true"
                     type="xsd:string"/>
        <xsd:element name="varStruct" nillable="true"
                     type="s:SimpleStruct"/>
    </xsd:sequence>
</xsd:complexType>
```

The preceding `StructWithNillables` schema type maps to the IDL struct, `StructWithNillables`, which uses generated nillable types, `float_nil`, `long_nil`, `string_nil` and `SimpleStruct_nil`, to represent the types of its member elements:

```
// IDL
union float_nil switch(boolean) {
    case TRUE:
        float value;
};
union long_nil switch(boolean) {
    case TRUE:
        long value;
};
union string_nil switch(boolean) {
    case TRUE:
        string value;
};

struct SimpleStruct {
    string_nil varAttrString;
    float varFloat;
    long varInt;
    string varString;
};
union SimpleStruct_nil switch(boolean) {
    case TRUE:
        SimpleStruct value;
};
```

```
struct StructWithNillables {
    float_nil varFloat;
    long_nil varInt;
    string_nil varString;
    SimpleStruct_nil varStruct;
};
```

CHAPTER 8

Security Interoperability

When a secure SOAP client interoperates with a secure CORBA server, it is often necessary to transform the SOAP client's credentials (for example, a username and password embedded in the SOAP header or in the HTTP header) into a form of credentials that the CORBA server understands. This chapter describes two different scenarios for propagating credentials between a SOAP client and a CORBA server.

In this chapter

This chapter discusses the following topics:

SOAP-to-CORBA Scenario

Overview

This section describes how to integrate a secure SOAP client with a secure CORBA server, by interposing a suitably configured SOAP-to-CORBA Artix router. The router transforms the SOAP client's WSSE username and password credentials into CSI/GSSUP credentials for the CORBA server.

In this section

This section contains the following subsections:

Overview of the Secure SOAP-to-CORBA Scenario

Overview

This subsection describes a secure SOAP-to-CORBA scenario, where the router is configured to integrate SOAP security with CORBA security. The key functionality provided by the router in this scenario is the ability to extract SOAP credentials (provided in the form of a WSSE username and password) and propagate them as CORBA-compatible GSSUP credentials.

SOAP-to-CORBA scenario

Figure 17 shows the outline of a scenario where WSSE username and password credentials, embedded in a SOAP header, are transformed into GSSUP credentials, embedded in a GIOP service context.

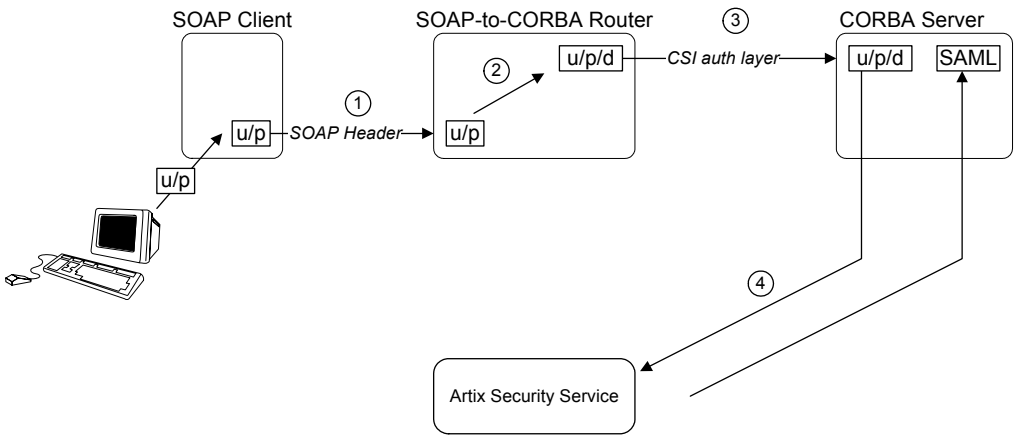

Figure 1: *Propagating Credentials Across a SOAP-to-CORBA Router*

Steps

The steps for propagating credentials across the SOAP-to-CORBA router, as shown in Figure 17, can be described as follows:

Stage	Description
1	The client initializes the WSSE username and password credentials, `u/p`, and sends these credentials, embedded in a WSSE SOAP header, across to the router.
2	The router extracts the received WSSE username and password credentials, `u/p`, and transfers them into GSSUP credentials, consisting of username, password and domain, `u/p/d`. The username and password are copied straight into the GSSUP credentials. The domain is set to a blank string (which acts as a wildcard that matches any domain).
3	The GSSUP credentials, `u/p/d`, are sent on to the CORBA server using the CSI authentication over transport mechanism.
4	The CORBA server authenticates the received GSSUP credentials, `u/p/d`, by calling out to the Artix security service (this step is performed automatically by the `gsp` plug-in).

Demonstration code

Demonstration code for this SOAP-to-CORBA scenario is available from the following location:

ArtixInstallDir/artix/*Version*/demos/security/secure_soap_corba

Enabling GSSUP propagation

To enable GSSUP propagation (where received username and password credentials are inserted into the outgoing GSSUP credentials by the router), set the following router configuration variable to `true`:

```
policies:bindings:corba:gssup_propagation = "true";
```

SOAP Client

Overview

When making an invocation, the SOAP client sends username and password credentials in a SOAP header (formatted according to the WSSE standard). This section describes how to program and configure a SOAP client to send WSSE username and password credentials.

Choice of credentials

In this example, the SOAP client is programmed to send username/password credentials in the SOAP header. It is also possible, however, to send username/password credentials in the HTTP header, using the HTTP Basic Authentication mechanism. The propagation mechanism in the router supports either type of credentials.

Setting the WSSE username and password

Example 36 shows how you can program a SOAP client to send username and password credentials using the WSSE standard.

Example 1: *SOAP Client Setting WSSE Username/Password Credentials*

```
// C++
...
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/context_constants.h>
#include <it_bus_pdk/context_attrs/bus_security_xsdTypes.h>
...
#include "HelloWorldClient.h"

IT_USING_NAMESPACE_STD

using namespace HW;
using namespace IT_Bus;
using namespace IT_ContextAttributes;

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

1       Bus* bus = Bus::create_reference();
        ContextRegistry* registry = bus->get_context_registry();
```

Example 1: *SOAP Client Setting WSSE Username/Password Credentials*

```
        ContextCurrent& current = registry->get_current();
        ContextContainer* request_contexts =
                              current.request_contexts();

        HelloWorldClient client;
        BusSecurity* security_attr;
        String* username;
        String* token;
        String string_out;
2       AnyType* output_attr = request_contexts->get_context(
            SECURITY_SERVER_CONTEXT,
            true
        );

3       security_attr = dynamic_cast<BusSecurity*> (output_attr);
4       security_attr->setWSSEUsernameToken("user_test");
        security_attr->setWSSEPasswordToken("user_password");
5       client.sayHi(string_out);
        ...
    }
    catch(IT_Bus::Exception& e)
    {
        ... // Handle exception (not shown)
        return -1;
    }
    return 0;
}
```

The preceding client code can be explained as follows:

1. The following four lines contain the standard steps for obtaining a pointer to the request context container object, `request_contexts`. The request context container object contains a collection of context objects, which contain various settings that can influence the next invocation request.

   For more details about Artix contexts, see the contexts chapter from *Developing Artix Applications in C++*.

2. Obtain a pointer to the `BusSecurity` context object from the request context container. The `BusSecurity` context is selected by passing the QName constant, `IT_ContextAttributes::SECURITY_SERVER_CONTEXT`,

as the first parameter to `get_context()`. The second parameter to `get_context()`, with the boolean value `true`, indicates that a new `BusSecurity` instance should be created, if one does not already exist.

3. Cast the return value from `get_context()` to the `IT_ContextAttributes::BusSecurity` type.
4. Call the `setWSSEUsernameToken()` and `setWSSEPasswordToken()` functions to specify the credentials to send with the next invocation. In this example the username and password are sent in the SOAP header and formatted according to the WSSE standard.
5. Invoke the remote WSDL operation, `sayHi`. The specified username and password are propagated in the SOAP header along with this invocation request.

Client configuration

Example 37 shows the configuration of the SOAP client in this scenario, which uses the `secure_artix.secure_soap_corba.client.gssup` configuration scope.

Example 2: *SOAP Client Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:iiops:1.2@localhost:58482,it_iiops:1.2@localhost:58
   482/IT_SecurityService";

      client
      {
          # Secure HTTPS client-side configuration
1         policies:https:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/t
   rusted_ca_lists/ca_list1.pem";
2       policies:https:client_secure_invocation_policy:requires
   = ["Confidentiality", "EstablishTrustInTarget"];
        policies:https:client_secure_invocation_policy:supports
   = ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];
```

Example 2: *SOAP Client Configuration*

```
3           principal_sponsor:use_principal_sponsor = "true";
            principal_sponsor:auth_method_id = "pkcs12_file";
            principal_sponsor:auth_method_data =
    ["filename=C:\artix_30/artix/3.0/demos/security/certificates/
    openssl/x509/certs/testaspen.p12", "password=testaspen"];
            ...
            gssup
            {
4               orb_plugins = ["xmlfile_log_stream", "https"];
            };
        };
};
```

The preceding client configuration can be explained as follows:

1. The trusted CA list policy specifies a listed of trusted CA certificates. During the SSL handshake, the client checks that the server's certificate is signed by one of the CA certificates from this list.
2. The client's HTTPS security policies require that connections are secure and the server identifies itself by sending an X.509 certificate.
3. Because this client supports mutual SSL authentication, the principal sponsor settings are used to associate an X.509 certificate with the client application.
4. The `https` plug-in *must* appear explicitly in the `orb_plugins` list in order to support the secure HTTPS protocol. Other required plug-ins (for example, `soap` and `at_http`) are loaded dynamically, based on the settings that appear in the client's WSDL contract.

SOAP-to-CORBA Router

Overview

The SOAP-to-CORBA router receives incoming SOAP/HTTP requests, translates them into IIOP requests and then forwards them on to a CORBA server. In addition to translating requests, the router is also configured to transfer the incoming username/password credentials (embedded in a SOAP header) into outgoing CSI credentials (embedded in a GIOP service context). Hence, the SOAP-to-CORBA router enables interoperation of SOAP/HTTP security with CORBA security.

Transferring credentials from SOAP to CORBA

The transferal of credentials from SOAP to CORBA obeys the following semantics:

- Extracting username/password credentials—the router can extract either WSSE username/password from the SOAP header or username/password from the HTTP header. If username/password credentials are sent in both headers, you can influence the priority by setting the `plugins:asp:security_level` configuration variable to one of the following values:
  - `REQUEST_LEVEL`—give priority to the WSSE username and password from the SOAP header.
  - `MESSAGE_LEVEL`—give priority to the username and password from the HTTP header.
- The username and password credentials are inserted into GSSUP credentials, which are transmitted using the CSI authentication over transport mechanism.
- The domain name in the GSSUP credentials is set to an empty string (which acts as a wildcard that matches any domain).
- The router does *not* attempt to authenticate the GSSUP credentials. Hence, the router does not call the Artix security service.
- The GSSUP credentials are used for a *single* invocation only.

Note: Internally, the GSSUP credentials are set using the `IT_CSI::CSICurrent3::set_effective_own_gssup_credentials_info()` function.

Router WSDL contract

Example 38 shows the WSDL contract for the SOAP-to-CORBA router.

Example 3: *SOAP-to-CORBA Router WSDL Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorldService"
   targetNamespace="http://xmlbus.com/HelloWorld"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:corba="http://schemas.iona.com/bindings/corba"
    ...
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    ...>
    <types>
        ...
    </types>
    ...
    <portType name="HelloWorldPortType">
        ...
    </portType>

    <binding name="HelloWorldPortBinding"
             type="tns:HelloWorldPortType">
        ...
    </binding>

    <binding name="CORBAHelloWorldBinding"
             type="tns:HelloWorldPortType">
        ...
    </binding>

1   <service name="HelloWorldService">
        <port binding="tns:HelloWorldPortBinding"
              name="HelloWorldPort">
2           <soap:address location="https://localhost:8085"/>
        </port>
    </service>

3   <service name="CORBAHelloWorldService">
        <port binding="tns:CORBAHelloWorldBinding"
              name="CORBAHelloWorldPort">
4           <corba:address
             location="file:../../corba/server/HelloWorld.ior"/>
            <corba:policy/>
        </port>
    </service>
```

Example 3: *SOAP-to-CORBA Router WSDL Contract*

```
5       <ns2:route name="r1">
            <ns2:source      port="HelloWorldPort"
                             service="tns:HelloWorldService"/>
            <ns2:destination port="CORBAHelloWorldPort"
                             service="tns:CORBAHelloWorldService"/>
        </ns2:route>
    </definitions>
```

The preceding router WSDL contract can be explained as follows:

1. The `HellowWorldService` specifies a SOAP/HTTP endpoint for the `HelloWorldPortType` port type.
2. The SOAP/HTTP endpoint has the address, `https://localhost:8085` (you might want to change this to specify the actual name of the host where the router is running).

   Note: The secure HTTPS protocol is used here (as indicated by the `https` prefix in the URL).

3. The `CORBAHelloWorldService` specifies a CORBA endpoint for the `HelloWorldPortType` port type.
4. The location of the CORBA endpoint is given by a stringified interoperable object reference (IOR), which is stored in the file, `HelloWorld.ior`. The CORBA server is programmed to create this file as it starts up.

   Note: A more sophisticated alternative for specifying the CORBA endpoint would be to use the CORBA Naming Service.

5. The `route` element sets up a route as follows:
   - The source endpoint (which receives incoming requests) is the SOAP/HTTP endpoint, `HelloWorldPort`.
   - The destination endpoint (to which the router sends outgoing requests) is the CORBA endpoint, `CORBAHelloWorldPort`.

Router configuration

Example 39 shows the configuration of the router in this scenario, which uses the `secure_artix.secure_soap_corba.switch.gssup` configuration scope.

Example 4: *SOAP-to-CORBA Router Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:iiops:1.2@localhost:58482,it_iiops:1.2@localhost:58
   482/IT_SecurityService";

      switch
      {
          ###################################
          # required for token propagation
          plugins:gsp:accept_asserted_authorization_info =
   "false";

          plugins:gsp:assert_authorization_info = "false";

          # iiop_tls config
1  policies:iiop_tls:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/t
   rusted_ca_lists/ca_list1.pem";
2  policies:iiop_tls:client_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering"];
   policies:iiop_tls:client_secure_invocation_policy:supports =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];
3         principal_sponsor:use_principal_sponsor = "true";
          principal_sponsor:auth_method_id = "pkcs12_file";
          principal_sponsor:auth_method_data =
   ["filename=router_cert.p12","password_file=router_cert.pwf"];

          # csi auth config
        policies:csi:auth_over_transport:authentication_service
   = "com.iona.corba.security.csi.AuthenticationService";
```

Example 4: *SOAP-to-CORBA Router Configuration*

```
4           policies:csi:auth_over_transport:client_supports =
    ["EstablishTrustInClient"];
            policies:csi:attribute_service:client_supports =
    ["IdentityAssertion"];

            #binding/plugin list
5           orb_plugins = ["xmlfile_log_stream", "iiop_profile",
    "giop", "iiop_tls", "routing", "gsp", "https"];
            binding:client_binding_list = ["OTS+POA_Coloc",
    "POA_Coloc", "CSI+OTS+GIOP+IIOP", "CSI+GIOP+IIOP",
    "CSI+OTS+GIOP+IIOP_TLS", "CSI+GIOP+IIOP_TLS","OTS+GIOP+IIOP",
    "GIOP+IIOP", "GIOP+IIOP_TLS"];

6           plugins:routing:wsdl_url="../../etc/router.wsdl";

            # Secure HTTPS server-side settings
            policies:https:trusted_ca_list_policy =
    "C:\artix_30/artix/3.0/demos/security/certificates/openssl/x5
    09/ca/cacert.pem";
          policies:https:target_secure_invocation_policy:requires
    = ["Confidentiality", "EstablishTrustInClient"];
          policies:https:target_secure_invocation_policy:supports
    = ["Confidentiality", "Integrity", "DetectReplay",
    "DetectMisordering", "EstablishTrustInClient",
    "EstablishTrustInTarget"];

            gssup
            {
                ################################################
                # flags to control credential propagation
7               policies:bindings:corba:token_propagation="false";
8               policies:bindings:corba:gssup_propagation="true";
                ################################################
            };
        };
    };
};
```

The preceding router configuration can be explained as follows:

1. This trusted CA list policy specifies the CA certificates that are used to check certificates received from the CORBA server during the SSL/TLS handshake.
2. This policy specifies that the router can only open secure IIOP/TLS connections to CORBA servers.
3. The principal sponsor settings associate an X.509 certificate with the Artix router.
4. CSI provides two different mechanisms for transporting credentials, both of which are supported by the router:
   - *Authorization over transport*—transfers credentials in the form of a username, password and domain name. This is the mechanism used in the current scenario.
   - *Identity assertion*—transfers credentials in the form of an asserted identity. This is the mechanism that is used in combination with single sign-on—see "Single Sign-On SOAP-to-CORBA Scenario" on page 154.
5. The `https` plug-in must be included in the `orb_plugins` list to enable the secure HTTPS protocol. The `iiop_tls` plug-in enables secure IIOP/TLS communication.

   Note: If you intend to use secure IIOP/TLS communications, it is best to remove the `iiop` plug-in from the `orb_plugins` list. This helps to safeguard against accidental mis-configuration.

6. This line specifies the location of the router WSDL contract.
7. The token propagation option is disabled in this scenario.
8. The GSSUP propagation option is enabled in this scenario. This is the key setting for enabling security interoperability. The CORBA binding extracts the username and password credentials from incoming SOAP/HTTP invocations and inserts them into an outgoing GSSUP credentials object, to be transmitted using CSI authentication over transport. The domain name in the outgoing GSSUP credentials is set to a blank string.

CORBA Server

Overview

In this scenario, the CORBA server must be configured to accept GSSUP credentials through the CSI authentication over transport mechanism. This subsection describes how to configure the CORBA server to authenticate the received CSI credentials.

Server configuration

Example 40 shows the configuration of the CORBA server in this scenario, which uses the `secure_artix.secure_soap_corba.server.gssup` configuration scope.

Example 5: *CORBA Server Supporting GSSUP Credentials*

```
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:iiops:1.2@localhost:58482,it_iiops:1.2@localhost:58
   482/IT_SecurityService";

     server
     {

         # binding/plugin list
         orb_plugins = ["local_log_stream", "iiop_profile",
   "giop", "iiop_tls", "gsp"];
         binding:client_binding_list = ["GIOP+EGMIOP",
   "OTS+POA_Coloc", "POA_Coloc", "OTS+TLS_Coloc+POA_Coloc",
   "TLS_Coloc+POA_Coloc", "GIOP+SHMIOP", "CSI+OTS+GIOP+IIOP",
   "CSI+GIOP+IIOP", "CSI+OTS+GIOP+IIOP_TLS",
   "CSI+GIOP+IIOP_TLS", "GIOP+IIOP", "GIOP+IIOP_TLS"];
       binding:server_binding_list = ["CSI+GSP", "CSI", "GSP"];

         # disable authorization
1        plugins:gsp:enable_authorization="false";

2        # disable client side caching
         # plugins:gsp:authentication_cache_size = "-1";
         # plugins:gsp:authentication_cache_timeout = "0";

         # csi auth config
```

Example 5: *CORBA Server Supporting GSSUP Credentials*

```
        policies:csi:auth_over_transport:authentication_service
   = "com.iona.corba.security.csi.AuthenticationService";
3        policies:csi:auth_over_transport:server_domain_name =
   "PCGROUP";
         policies:csi:attribute_service:target_supports =
   ["IdentityAssertion"];


         # iiop_tls config
policies:iiop_tls:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/t
   rusted_ca_lists/ca_list1.pem";
4  policies:iiop_tls:target_secure_invocation_policy:supports =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInTarget",
   "EstablishTrustInClient"];
policies:iiop_tls:target_secure_invocation_policy:requires =
   ["Integrity", "Confidentiality", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient"];
5        principal_sponsor:use_principal_sponsor = "true";
         principal_sponsor:auth_method_id = "pkcs12_file";
         principal_sponsor:auth_method_data =
   ["filename=server_cert.p12","password_file=server_cert.pwf"];

         # Configuration required for Token propagation.
6        plugins:gsp:accept_asserted_authorization_info =
   "false";

         # Configuration required for GSSUP propagation.
7        policies:csi:auth_over_transport:target_requires =
   ["EstablishTrustInClient"];
         policies:csi:auth_over_transport:target_supports =
   ["EstablishTrustInClient"];
      };
   };
};
```

The preceding server configuration can be described as follows:

1. In this example, authorization is disabled for simplicity. You can enable authorization, however, if your application requires it.
2. You might want to disable client side caching for testing purposes (this would force the server to contact the security service with every invocation). Normally, however, you should leave these lines commented out, as shown here. Client caching improves performance considerably.
3. If needed for authorization purposes, you can set the domain name here.
4. These settings for the IIOP/TLS target secure invocation policy ensure that the server accepts only secure connections. The server also requires the `EstablishTrustInClient` association option, which implies that clients must provide an X.509 certificate during the SSL/TLS handshake.
5. The principal sponsor settings associate an X.509 certificate (in PKCS#12 format) with the CORBA server.
6. If the server receives credentials in the form of an SSO token, this setting ensures that the server re-authenticates the token, instead of relying on SAML data propagated with the request.
7. These CSI authorization over transport policies require clients to provide GSSUP credentials, which contain a username, password and domain name. The gsp plug-in is then responsible for contacting the Artix security service to authenticate these credentials.

Single Sign-On SOAP-to-CORBA Scenario

Overview

This section describes how to integrate a single sign-on SOAP client with a secure CORBA server, by interposing a suitably configured SOAP-to-CORBA Artix router.

In this section

This section contains the following subsections:

Overview of the Secure SSO SOAP-to-CORBA Scenario

Overview

This subsection describes a variation of the secure SOAP-to-CORBA scenario, where the client is configured to use *single sign-on* (SSO). In this scenario, the client authenticates the username and password with the login service prior to sending an invocation to the router. Instead of sending username and password credentials to the router, the client sends the SSO token it received from the login service. The router can then be configured to propagate the SSO token to the remote CORBA server.

SSO SOAP-to-CORBA scenario

Figure 18 shows the outline of a scenario where an SSO token, embedded in a SOAP header, is transformed into a CSI identity token, embedded in a GIOP header (GIOP service context).

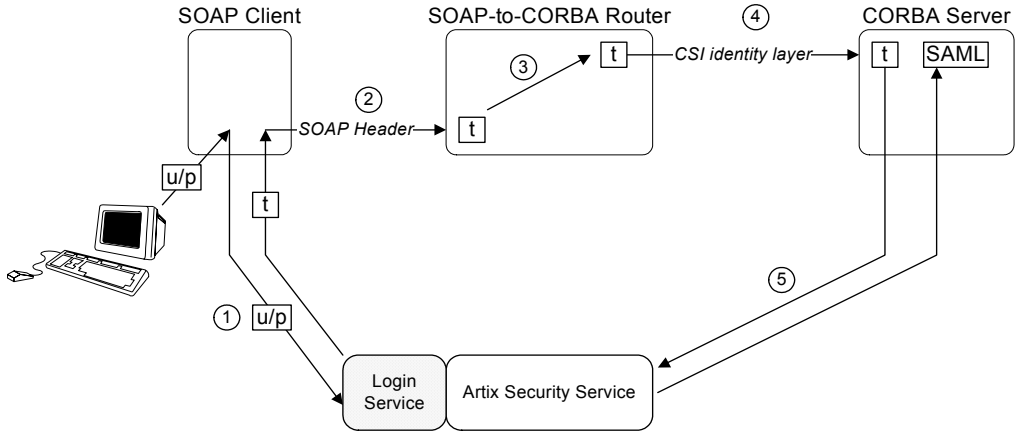

Figure 2: *Propagating an SSO Token Across a SOAP-to-CORBA Router*

Steps

The steps for propagating credentials across the SOAP-to-CORBA router, as shown in Figure 17, can be described as follows:

Stage	Description
1	When single sign-on is enabled, the client calls out to the login service, passing in the client's WSSE credentials, `u/p`, in order to obtain an SSO token.
2	When the client invokes an operation on the router, the SSO token, t, is sent as the password in the WSSE credentials.
3	The router extracts the SSO token, `t`, from the received WSSE credentials and then inserts the SSO token into the outgoing CSI identity token. **Note:** The router should not attempt to authenticate the received SSO token. In the current example, authentication does not occur, because the router does not load the `artix_security` plug-in.
4	The SSO token, `t`, is sent on to the CORBA server using the CSI identity assertion mechanism.
5	The CORBA server re-authenticates the client's SSO token, `t`, by calling out to the Artix security service. The return value contains the SAML role and realm data for the token.

Demonstration code

Demonstration code for the SSO SOAP-to-CORBA scenario is available from the following location:

ArtixInstallDir/artix/*Version*/demos/security/secure_soap_corba

Enabling token propagation

To enable SSO token propagation (where received SSO tokens are inserted into the outgoing CSI identity token by the router), set the following router configuration variable to `true`:

```
policies:bindings:corba:token_propagation = "true";
```

SSO SOAP Client

Overview

This subsection describes how to configure a SOAP client to use single sign-on. The initial client credentials are a WSSE username and password (programmed as shown in "Setting the WSSE username and password" on page 141). After contacting the login service, however, the client uses an SSO token as its credentials for subsequent invocations.

SSO client configuration

Example 41 shows the configuration of the single sign-on SOAP client, which uses the `secure_artix.secure_soap_corba.client.token` configuration scope.

Example 6: *Single Sign-On SOAP Client Configuration*

```
# Artix Configuration File
...
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:iiops:1.2@localhost:58482,it_iiops:1.2@localhost:58
   482/IT_SecurityService";

      client
      {
          # Secure HTTPS client-side configuration
          policies:https:trusted_ca_list_policy =
   "C:\artix_30/artix/3.0/demos/security/certificates/tls/x509/t
   rusted_ca_lists/ca_list1.pem";
        policies:https:client_secure_invocation_policy:requires
   = ["Confidentiality", "EstablishTrustInTarget"];
        policies:https:client_secure_invocation_policy:supports
   = ["Confidentiality", "Integrity", "DetectReplay",
   "DetectMisordering", "EstablishTrustInClient",
   "EstablishTrustInTarget"];

          principal_sponsor:use_principal_sponsor = "true";
          principal_sponsor:auth_method_id = "pkcs12_file";
          principal_sponsor:auth_method_data =
   ["filename=C:\artix_30/artix/3.0/demos/security/certificates/
   openssl/x509/certs/testaspen.p12", "password=testaspen"];
```

Example 6: *Single Sign-On SOAP Client Configuration*

```
          ...
          token
          {
1             orb_plugins = ["xmlfile_log_stream",
   "login_client", "https"];
2             binding:artix:client_request_interceptor_list=
   "login_client";
3             bus:initial_contract:url:login_service =
   "../../wsdl/login_service.wsdl";
4             plugins:login_client:shlib_name =
   "it_login_client";
          };
      };
};
```

The preceding configuration can be explained as follows:

1. To enable the single sign-on functionality in the client, add the `login_client` plug-in to the list of ORB plug-ins.
2. It is also necessary to add `login_client` to the Artix client request interceptor list (the single sign-on functionality is implemented by a client request interceptor).
3. The `bus:initial_contract:url:login_service` variable specifies the location of the login service's WSDL contract. This contract contains the address of the login service endpoint.
4. This line identifies the dynamic library that contains the `login_client` plug-in.

SSO SOAP-to-CORBA Router

Overview

The single sign-on SOAP-to-CORBA router is configured similarly to the normal SOAP-to-CORBA router ("SOAP-to-CORBA Router" on page 145), except that the CORBA binding is configured to enable token propagation instead of GSSUP propagation.

Transferring credentials from SOAP to CORBA

The transferal of credentials from SOAP to CORBA in the single sign-on scenario obeys the following semantics:

- The SSO token credentials are inserted into a CSI identity token, which is transmitted using the CSI identity assertion mechanism.
- The router does *not* attempt to authenticate the SSO token. Hence, the router does not call the Artix security service.
- The SSO token is used for a *single* invocation only.

Note: Internally, the CSI identity token is set using the `IT_CSI::CSICurrent2::set_received_itt_principal_name_identity_token()` function.

SSO router configuration

Example 42 shows the configuration of the single sign-on router, which uses the `secure_artix.secure_soap_corba.switch.token` configuration scope.

Example 7: *Single Sign-On SOAP-to-CORBA Router Configuration*

```
# Artix Configuration File
secure_artix
{
   secure_soap_corba
   {
     initial_references:IT_SecurityService:reference =
   "corbaloc:iiops:1.2@localhost:58482,it_iiops:1.2@localhost:58
   482/IT_SecurityService";


      switch
      {
          plugins:gsp:assert_authorization_info = "false";
```

Example 7: *Single Sign-On SOAP-to-CORBA Router Configuration*

```
1           # Common configuration
            ...
2           policies:csi:attribute_service:client_supports =
    ["IdentityAssertion"];
            ...
            token
            {
3               policies:bindings:corba:token_propagation="true";
4               policies:bindings:corba:gssup_propagation="false";
            };
            ...
       };
    };
};
```

The preceding router configuration can be explained as follows:

1. The rest of the `secure_artix.secure_soap_corba.switch` scope is the same as the scenario without single sign-on. See "SOAP-to-CORBA Router" on page 145 for details.
2. This line is of particular importance for the single sign-on scenario. It enables the CSI identity assertion mechanism, which is needed to transmit the SSO token to the CORBA server.
3. The token propagation option is enabled in this scenario. This is the key setting for enabling security interoperability. The CORBA binding extracts the SSO token from incoming SOAP/HTTP invocations and inserts the token into an outgoing IIOP request, to be transmitted using CSI identity assertion.
4. The GSSUP propagation option is disabled in this scenario.

CHAPTER 9

Monitoring GIOP Message Content

Artix includes the GIOP Snoop tool for intercepting and displaying GIOP message content.

In this chapter

This chapter contains the following sections:

Introduction to GIOP Snoop

Overview

GIOP Snoop is a GIOP protocol level plug-in for intercepting and displaying GIOP message content. This plug-in implements message level interceptors that can participate in client and/or server side bindings over any GIOP-based transport. The primary purposes of GIOP Snoop are to provide a protocol level monitor and debug aid.

GIOP plug-ins

The primary protocol for inter-ORB communications is the General Inter-ORB Protocol (GIOP) as defined the CORBA Specification.

Configuring GIOP Snoop

Overview

GIOP Snoop can be configured for debugging in client, server, or both depending on configuration. This section includes the following configuration topics:

- Loading the GIOP Snoop plug-in.
- Client-side snooping.
- Server-side snooping.
- GIOP Snoop verbosity levels.
- Directing output to a file.

Loading the GIOP Snoop plug-in

For either client or server configuration, the GIOP Snoop plug-in must be included in the Orbix `orb_plugins` list (`...` denotes existing configured settings):

```
orb_plugins = [..., "giop_snoop", ...];
```

In addition, the `giop_snoop` plug-in must be located and loaded using the following settings:

```
# Artix Configuration File
plugins:giop_snoop:shlib_name = "it_giop_snoop";
```

Client-side snooping

To enable client-side snooping, include the `GIOP_SNOOP` factory in the client binding list. In this example, GIOP Snoop is enabled for IIOP-specific bindings:

```
binding:client_binding_list =
      [..., "GIOP+GIOP_SNOOP+IIOP", ...];
```

Server-side snooping

To enable server-side snooping, include the `GIOP_SNOOP` factory in the server binding list.

```
plugins:giop:message_server_binding_list =
     [..., "GIOP_SNOOP+GIOP", ...];
```

GIOP Snoop verbosity levels

You can use the following variable to control the GIOP Snoop verbosity level:

```
plugins:giop_snoop:verbosity = "1";
```

The verbosity levels are as follows:

1	LOW
2	MEDIUM
3	HIGH
4	VERY HIGH

These verbosity levels are explained with examples in "GIOP Snoop Output" on page 166.

Directing output to a file

By default, output is directed to standard error (`stderr`). However, you can specify an output file using the following configuration variable:

```
plugins:giop_snoop:filename = "<some-file-path>";
```

A month/day/year time stamp is included in the output filename with the following general format:

```
<filename>.MMDDYYYY
```

As a result, for a long running application, each day results in the creation of a new log file. To enable administrators to control the size and content of output files GIOP Snoop does not hold output files open. Instead, it opens and then closes the file for each snoop message trace. This setting is enabled with:

```
plugins:giop_snoop:rolling_file = "true";
```

GIOP Snoop Output

Overview

The output shown in this section uses a simple example that shows client-side output for a single binding and operation invocation. The client establishes a client-side binding that involves a message interceptor chain consisting of IIOP, GIOP Snoop, and GIOP. The client then connects to the server and first sends a `[LocateRequest]` to the server to test if the target object is reachable. When confirmed, a two-way invocation `[Request]` is sent, and the server processes the request. When complete, the server sends a `[Reply]` message back to the client.

Output detail varies depending on the configured verbosity level. With level `1` (`LOW`), only basic message type, direction, operation name and some GIOP header information (version, and so on) is given. More detailed output is possible, as described under the following examples.

LOW verbosity client-side snooping

An example of `LOW` verbosity output is as follows:

```
[Conn:1] Out:(first for binding)  [LocateRequest] MsgLen: 39 ReqId: 0
[Conn:1] In: (first for binding)  [LocateReply] MsgLen: 8 ReqId: 0
    Locate status: OBJECT_HERE
[Conn:1] Out:  [Request] MsgLen: 60 ReqId: 1  (two-way)
    Operation (len 8) 'null_op'
[Conn:1] In:   [Reply] MsgLen: 12 ReqId: 1
    Reply status (0) NO_EXCEPTION
```

This example shows an initial conversation from the client-side perspective. The client transmits a `[LocateRequest]` message to which it receives a `[LocateReply]` indicates that the server supports the target object. It then makes an invocation on the operation `null_op`.

The `Conn` indicates the logical connection. Because GIOP may be mapped to multiple transports, there is no transport specific information visible to interceptors above the transport (such as file descriptors) so each connection is given a logical identifier. The first incoming and outgoing GIOP message to pass through each connection are indicated by `(first for binding)`.

The direction of the message is given (`Out` for outgoing, `In` for incoming), followed by the GIOP and message header contents. Specific information includes the GIOP version (version 1.2 above), message length and a unique request identifier (`ReqId`), which associates `[LocateRequest]` messages with their corresponding `[LocateReply]` messages. The `(two-way)` indicates the operation is two way and a response (`Reply`) is expected. String lengths such as `len 8` specified for `Operation` includes the trailing null.

MEDIUM verbosity client-side snooping

An example of `MEDIUM` verbosity output is as follows:

```
16:24:39 [Conn:1] Out:(first for binding)  [LocateRequest]   GIOP v1.2  MsgLen: 39
   Endian: big  ReqId: 0
   Target Address (0: KeyAddr)
   ObjKey (len 27) ':>.11........\..A..........'

16:24:39 [Conn:1] In: (first for binding)  [LocateReply]     GIOP v1.2  MsgLen: 8
   Endian: big  ReqId: 0
   Locate status: OBJECT_HERE

16:24:39 [Conn:1] Out:  [Request]          GIOP v1.2  MsgLen: 60
   Endian: big  ReqId: 1  (two-way)
   Target Address (0: KeyAddr)
   ObjKey (len 27) ':>.11........\..A..........'
   Operation (len 8) 'null_op'

16:24:39 [Conn:1] In:   [Reply]            GIOP v1.2  MsgLen: 12
   Endian: big  ReqId: 1
   Reply status (0) NO_EXCEPTION
```

For `MEDIUM` verbosity output, extra information is provided. The addition of time stamps (in *`hh:mm:ss`*) precedes each snoop line. The byte order of the data is indicated (`Endian`) along with more detailed header information such as the target address shown in this example. The target address is a GIOP 1.2 addition in place of the previous object key data.

HIGH verbosity client side snooping

The following is an example of `HIGH` verbosity output:

```
16:24:39 [Conn:1] Out:(first for binding)  [LocateRequest]    GIOP v1.2  MsgLen: 39
    Endian: big  ReqId: 0
    Target Address (0: KeyAddr)
      ObjKey (len 27) ':>.11...........A..........'
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][03][00][00][00][27]
    Msg Hdr  (len 39): [00][00][00][00][00][00][00][00][00][00][00][1b][3a][3e]
[02][31][31][0c][00][00][00][00][00][00][0f][05][00][00][41][c6][08][00][00][00]
[00][00][00][00][00]
[---- end of message ----]

16:31:37 [Conn:1] In: (first for binding)  [LocateReply]      GIOP v1.2  MsgLen: 8
    Endian: big  ReqId: 0
    Locate status: OBJECT_HERE
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][04][00][00][00][08]
    Msg Hdr  (len 8): [00][00][00][00][00][00][00][01]
[---- end of message ----]

16:31:37 [Conn:1] Out:  [Request]         GIOP v1.2  MsgLen: 60
    Endian: big  ReqId: 1  (two-way)
    Target Address (0: KeyAddr)
      ObjKey (len 27) ':>.11...........A..........'
    Operation (len 8) 'null_op'
    No. of Service Contexts: 0
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][00][00][00][00][3c]
    Msg Hdr  (len 60): [00][00][00][01][03][00][00][00][00][00][00][00][00][00]
[00][1b][3a][3e][02][31][31][0c][00][00][00][00][00][00][0f][05][00][00][41][c6]
[08][00][00][00][00][00][00][00][00][00][00][00][00][08][6e][75][6c][6c][5f][6f]
[70][00][00][00][00][00]
[---- end of message ----]

16:31:37 [Conn:1] In:   [Reply]           GIOP v1.2  MsgLen: 12
    Endian: big  ReqId: 1
    Reply status (0) NO_EXCEPTION
    No. of Service Contexts: 0
    GIOP Hdr (len 12): [47][49][4f][50][01][02][00][01][00][00][00][0c]
    Msg Hdr  (len 12): [00][00][00][01][00][00][00][00][00][00][00][00]
[---- end of message ----]
```

This level of verbosity includes all header data, such as service context data. ASCII-hex pairs of GIOP header and message header content are given to show the exact on-the-wire header values passing through the interceptor. Messages are also separated showing inter-message boundaries.

VERY HIGH verbosity client side snooping

This is the highest verbosity level available. Displayed data includes `HIGH` level output and in addition the message body content is displayed. Because the plug-in does not have access to IDL interface definitions, it does not know the data types contained in the body (parameter values, return values and so on) and simply provides ASCII-hex output. Body content display is truncated to a maximum of 4 KB with no output given for an empty body. Body content output follows the header output, for example:

```
...
GIOP Hdr (len 12): [47][49][4f][50][01][02][00][01][00][00][00][0c]
Msg Hdr  (len 12): [00][00][00][01][00][00][00][00][00][00][00][00]
Msg Body (len <x>): <content>
...
```

Configuring a CORBA Binding

CORBA bindings are described using a variety of IONA-specific WSDL elements within the WSDL binding element. In most cases, the CORBA binding description is generated automatically using the wsdltocorba utility. Usually, it is unnecessary to modify generated CORBA bindings.

Namespace

The WSDL extensions used to describe CORBA data mappings and CORBA transport details are defined in the WSDL namespace `http://schemas.iona.com/bindings/corba`. To use the CORBA extensions you will need to include the following in the `<definitions>` tag of your contract:

```
xmlns:corba="http://schemas.iona.com/bindings/corba"
```

corba:binding element

The `corba:binding` element indicates that the binding is a CORBA binding. This element has one required attribute: `repositoryID`. `repositoryID` specifies the full type ID of the interface. The type ID is embedded in the object's IOR and therefore must conform to the IDs that are generated from an IDL compiler. These are of the form:

```
IDL:module/interface:1.0
```

The `corba:binding` element also has an optional attribute, `bases`, that specifies that the interface being bound inherits from another interface. The value for `bases` is the type ID of the interface from which the bound interface inherits. For example, the following IDL:

```
//IDL
interface clash{};
interface bad : clash{};
```

would produce the following `corba:binding`:

```
<corba:binding repositoryID="IDL:bad:1.0"
               bases="IDL:clash:1.0"/>
```

corba:operation element

The `corba:operation` element is an IONA-specific element of `<operation>` and describes the parts of the operation's messages. `<corba:operation>` takes a single attribute, `name`, which duplicates the name given in `<operation>`.

corba:param element

The `corba:param` element is a member of `<corba:operation>`. Each `<part>` of the input and output messages specified in the logical operation, except for the part representing the return value of the operation, must have a

corresponding <corba:param>. The parameter order defined in the binding must match the order specified in the IDL definition of the operation. <corba:param> has the following required attributes:

mode	Specifies the direction of the parameter. The values directly correspond to the IDL directions: in, inout, out. Parameters set to in must be included in the input message of the logical operation. Parameters set to out must be included in the output message of the logical operation. Parameters set to inout must appear in both the input and output messages of the logical operation.
idltype	Specifies the IDL type of the parameter. The type names are prefaced with corba: for primitive IDL types, and corbatm: for complex data types, which are mapped out in the corba:typeMapping portion of the contract.
name	Specifies the name of the parameter as given in the logical portion of the contract.

corba:return element

The corba:return element is a member of <corba:operation> and specifies the return type, if any, of the operation. It only has two attributes:

name	Specifies the name of the parameter as given in the logical portion of the contract.
idltype	Specifies the IDL type of the parameter. The type names are prefaced with corba: for primitive IDL types and corbatm: for complex data types which are mapped out in the corba:typeMapping portion of the contract.

corba:raises element

The corba:raises element is a member of <corba:operation> and describes any exceptions the operation can raise. The exceptions are defined as fault messages in the logical definition of the operation. Each fault message must have a corresponding corba:raises element. The corba:raises element has one required attribute, exception, which specifies the type of data returned in the exception.

In addition to operations specified in <corba:operation> tags, within the <operation> block, each <operation> in the binding must also specify empty input and output elements as required by the WSDL specification. The CORBA binding specification, however, does not use them.

For each fault message defined in the logical description of the operation, a corresponding `fault` element must be provided in the `<operation>`, as required by the WSDL specification. The `name` attribute of the `fault` element specifies the name of the schema type representing the data passed in the fault message.

Example

For example, a logical interface for a system to retrieve employee information might look similar to `personalInfoLookup`, shown in Example 43.

Example 1: *personalInfo lookup port type*

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int" />
<message />
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo" />
<message />
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound" />
<message />
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest" />
    <output name="return" message="personalLookupResponse" />
    <fault name="exception" message="idNotFoundException" />
  </ operation>
</ portType>
```

The CORBA binding for `personalInfoLookup` is shown in Example 44.

Example 2: *personalInfoLookup CORBA Binding*

```
<binding name="personalInfoLookupBinding" type="tns:personalInfoLookup">
  <corba:binding repositoryID="IDL:personalInfoLookup:1.0"/>
  <operation name="lookup">
    <corba:operation name="lookup">
      <corba:param name="empId" mode="in" idltype="corba:long"/>
      <corba:return name="return" idltype="corbatm:personalInfo"/>
      <corba:raises exception="corbatm:idNotFound"/>
    </corba:operation>
    <input/>
    <output/>
    <fault name="personalInfoLookup.idNotFound"/>
  </operation>
</binding>
```

Configuring a CORBA Port

CORBA ports are described using the IONA-specific WSDL elements, corba:address and corba:policy, within the WSDL port element, to specify how a CORBA object is exposed.

Namespace

Example 45 shows the namespace entries you need to add to the `definitions` element of your contract to use the CORBA extensions.

Example 1: *Artix CORBA Extension Namespaces*

```
<definitions
  ...
  xmlns:iiop="http://schemas.iona.com/bindings/corba"
  ... >
```

corba:address element

The IOR of the CORBA object is specified using the `corba:address` element. You have four options for specifying IORs in Artix contracts:

- Specify the objects IOR directly, by entering the object's IOR directly into the contract using the stringified IOR format:

  ```
  IOR:22342....
  ```

- Specify a file location for the IOR, using the following syntax:

  ```
  file:///file_name
  ```

  Note: The file specification requires three backslashes (`///`).

  It is usually simplest to specify the file name using an absolute path. If you specify the file name using a relative path, the location is taken to be relative to the directory the Artix process is started in, *not* relative to the containing WSDL file.

- Specify that the IOR is published to a CORBA name service, by entering the object's name using the `corbaname` format:

  ```
  corbaname:rir/NameService#object_name
  ```

  For more information on using the name service with Artix see *Deploying and Managing Artix Solutions*.

- Specify the IOR using `corbaloc`, by specifying the port at which the service exposes itself, using the `corbaloc` syntax.

  ```
  corbaloc:iiop:host:port/service_name
  ```

When using `corbaloc`, you must be sure to configure your service to start up on the specified host and port.

---

corba:policy element

Using the optional `corba:policy` element, you can describe a number of POA polices the Artix service will use when creating the POA for connecting to a CORBA application. These policies include:

- POA Name.
- Persistence.
- ID Assignment.

Setting these policies lets you exploit some of the enterprise features of IONA's Orbix 6.x, such as load balancing and fault tolerance, when deploying an Artix integration project. For information on using these advanced CORBA features, see the Orbix documentation.

POA Name

By default, an Artix POA is created with the default name, `{`*`ServiceNamespace`*`}`*`ServiceLocalPart`*`#`*`PortName`*. For example, if a CORBA port is defined by the following WSDL fragment:

```
<definitions
  ...
  xmlns:corbatm="http://iona.com/mycorbaservice" >

    <service name="CorbaService">
        <port binding="corbatm:CorbaBinding" name="CorbaPort">
            <corba:address
                location="file:../../hello_world_service.ior"/>
        </port>
    </service>
    ...
```

The unique POA name automatically generated for this CORBA port is `{http://iona.com/mycorbaservice}CorbaService#CorbaPort`.

Alternatively, you can specify the POA name explicitly by setting the `poaname` attribute, as follows:

```
<corba:policy poaname="poa_name" />
```

When setting a POA name using the `poaname` attribute, it is your responsibility to ensure that the POA name is unique. That is, the POA name should *not* be shared between CORBA ports within a service or across CORBA services.

Persistence

By default Artix POA's have a persistence policy of `false`. To set the POA's persistence policy to `true`, use the following:

```
<corba:policy persistent="true" />
```

ID Assignment

By default Artix POAs are created with a `SYSTEM_ID` policy, meaning that their ID is assigned by the ORB. To specify that the POA connecting a specific object should use a user-assigned ID, use the following:

```
<corba:policy serviceid="POAid" />
```

This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Example

For example, a CORBA port for the `personalInfoLookup` binding would look similar to Example 46:

Example 2: *CORBA personalInfoLookup Port*

```
<service name="personalInfoLookupService">
  <port name="personalInfoLookupPort"
        binding="tns:personalInfoLookupBinding">
    <corba:address location="file:///objref.ior" />
    <corba:policy persistent="true" />
    <corba:policy serviceid="personalInfoLookup" />
  </ port>
</ service>
```

Artix expects the IOR for the CORBA object to be located in a file called `objref.ior` (relative to the directory in which the Artix process is started), and creates a persistent POA with an object id of `personalInfo` to connect the CORBA application.

APPENDIX 12

CORBA Utilities in Artix

Use the idltowsdl utility to convert OMG IDL to WSDL and use the wsdltocorba utility to generate CORBA bindings and to convert WSDL to OMG IDL.

In this chapter

This chapter discusses the following topics:

Generating a CORBA Binding

Overview

The `wsdltocorba` utility can perform two distinct tasks:

- Generate a CORBA binding.
- Convert WSDL to OMG IDL.

This section discusses how to use the `wsdltocorba` utility to add a CORBA binding to an existing WSDL contract.

WSDLTOCORBA

Synopsis

```
wsdltocorba -corba -i port-type [-d directory] [-o file]
[-props namespace] [-?] [-v] [-verbose] wsdl_file
```

Options

The command has the following options:

`-corba`	Instructs the tool to generate a CORBA binding for the specified port type.
`-i` *`port-type`*	Specifies the name of the port type being mapped to a CORBA binding.
`-d` *`directory`*	Specifies the directory into which the new WSDL file is written.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`wsdl_file`*`-corba.wsdl`.
`-props` *`namespace`*	Specifies the target namespace for the `corba:typeMapping` element (an element that defines the WSDL-to-IDL mappings for complex types).
`-?`	Display detailed information about the options.
`-v`	Display the version of the utility.
`-verbose`	Write a detailed log to standard output while the utility is running.

Converting WSDL to OMG IDL

Overview

The `wsdltocorba` utility can perform two distinct tasks:

- Generate a CORBA binding.
- Convert WSDL to OMG IDL.

This section discusses how to use the `wsdltocorba` utility to convert a WSDL contract into an OMG IDL file.

WSDLTOCORBA

Synopsis

```
wsdltocorba -idl -b binding [-d directory] [-o file] [-?] [-v]
[-verbose] wsdl_file
```

Options

The command has the following options:

`-idl`	Instructs the tool to generate an IDL file from the specified binding.
`-b binding`	Specifies the CORBA binding from which to generate IDL.
`-d directory`	Specifies the directory into which the new IDL file is written.
`-o file`	Specifies the name of the generated IDL file. Defaults to *wsdl_file*.idl.
`-?`	Display detailed information about the options.
`-v`	Display the version of the utility.
`-verbose`	Write a detailed log to standard output while the utility is running.

Converting OMG IDL to WSDL

Overview

IONA's IDL compiler supports several command line flags that specify how to create a WSDL file from an IDL file. The default behavior of the tool is to create WSDL file that uses wrapped doc/literal style messages. Wrapped doc/literal style messages have a single part, defined using an element that wraps all of the elements in the message.

IDLTOWSDL

Synopsis

```
idltowsdl [ -I idl-include-directory ]* [ -3 ] [ -o output-directory
] [ -a corba-address ] [ -b ] [ -f corba-address-file ] [ -n
schema-import-file ] [ -s idl-sequence-type ] [ -w target-namespace
] [ -x schema-namespace ] [ -t type-map-namespace ] [ -useTypes ]
[ -unwrap ] [ -r reference-schema-file ] [ -L logical-wsdl-file ]
[ -P physical-wsdl-file ] [ -T schema-file-name ] [ -fasttrack ] [
-interface interface-name ] [ -soapaddr soap-port-address ] [
-qualified ] [ -inline ] [ -e xml-encoding-type ] [ -? ] [ -v ] [
-verbose ] IDLFile
```

Options

The command has the following options:

`-I idl-include-directory`	Specify a directory to be included in the search path for the IDL preprocessor.
`-3`	Select parsing mode for compatibility with legacy Orbix 3 IDL files.
`-o output-directory`	Specifies the directory into which the WSDL file is written.
`-a corba-address`	Specifies an absolute address through which the object reference may be accessed. The *corba-address* may be a relative or absolute path to a file, or a corbaname URL
`-b`	Specifies that bounded strings are to be treated as unbounded. This eliminates the generation of the special types for the bounded string.

`-f` *`corba-address-file`*	Specifies a file containing a string representation of an object reference. The object reference is placed in the `corba:address` element in the `<port>` definition of the generated service. The *`corba-address-file`* must exist when you run the `idltowsdl` utility.
`-n` *`schema-import-file`*	Specifies that a schema file, *`schema-import-file`*, is to be included in the generated contract by an import statement. This option cannot be used with the `-T` option.
`-s` *`idl-sequence-type`*	Specifies the XML schema type used to map the IDL `sequence<octet>` type. Valid values are `base64Binary` or `hexBinary`. The default is `base64Binary`.
`-w` *`target-namespace`*	Specifies the namespace to use for the WSDL `targetNamespace`. The default is `http://schemas.iona.com/idl/`*`IDLFile`*.
`-x` *`schema-namespace`*	Specifies the namespace to use for the Schema `targetNamespace`. The default is `http://schemas.iona.com/idltypes/`*`IDLFile`*.
`-t` *`type-map-namespace`*	Specifies the namespace to use for the CORBA TypeMapping `targetNamespace`. The default is `http://schemas.iona.com/typemap/corba/`*`IDLFile`*.
`-useTypes`	Generate rpc style messages. rpc style messages have parts defined using XMLSchema types instead of XML elements.
`-unwrap`	Generate unwrapped doc/literal messages. Unwrapped messages have parts that represent individual elements. Unlike wrapped messages, unwrapped messages can have multiple parts and are not allowed by the WS-I.
`-r` *`reference-schema-file`*	Specify the pathname of the schema file imported to define the `Reference` type. If the `-r` option is not given, the idl compiler gets the schema file pathname from `ReferenceSchemaFile` setting in `etc/idl.cfg`.

`-L` *`logical-wsdl-file`*	Specifies that the logical portion of the generated WSDL specification into is written to *`logical-wsdl-file`*. The *`logical-wsdl-file`* is then imported into the default generated file.
`-P` *`physical-wsdl-filen`*	Specifies that the physical portion of the generated WSDL specification into is written to *`physical-wsdl-file`*. The *`physical-wsdl-file`* is then imported into the default generated file.
`-T` *`schema-file-name`*	Specifies that the schema types are to be generated into a separate file. The schema file is included in the generated contract using an import statement. This option cannot be used with the `-n` option.
`-fasttrack`	Provides a fast way of generating a router contract for a router that converts incoming SOAP/HTTP messages into CORBA invocations. The `-interface` option must always be specified when `-fasttrack` is used.
`-interface` *`interface-name`*	Used in combination with the `-fasttrack` option to specify the IDL interface that is exposed through the generated router contract.
`-soapaddr` *`soap-port-address`*	Used in combination with the `-fasttrack` option to specify the address of the generated SOAP port. The address is specified in the format *`Host`*`:`*`Port`*.
`-qualified`	Generate the schemas in the WSDL contract with the `elementFormDefault` and `attributeFormDefault` attributes set to `qualified`. This implies that elements and attributes appearing in instance documents must be explicitly qualified by a namespace.
`-inline`	Normally, when you specify a schema file using the `-n` option, the schema is imported by a generated `xsd:import` element, which sets the `schemaLocation` attribute. If you specify the `-inline` option, however, the schema is included directly in the generated WSDL contract and the generated `xsd:import` element omits the `schemaLocation` attribute.

`-e` *`xml-encoding-type`*	Use the specified WSDL encoding for the value of the `encoding` attribute in the generated `<?xml ... ?>` tag. The default is UTF-8.
`-?`	Display detailed information about the options.
`-v`	Display the version of the utility.
`-verbose`	Write a detailed log to standard output while the utility is running.

Note: 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.

Orbix 3 legacy compatibility

To address some issues associated with Orbix 3 migration, the Artix IDL compiler supports a `-3` option, which causes the following behavior in the `idltowsdl` utility:

- Case sensitivity is activated—this means that name lookup during parsing is case sensitive. While technically incorrect according to the CORBA specification, some legacy IDL files might require case sensitivity. The IDL compiler issues warnings, if case sensitivity rules are broken.
- New IDL keywords added since CORBA 2.3 (for example, `factory` and `local`) are treated as ordinary identifiers, but warnings are issued.
- If a different spelling of the keyword `Object` is encountered (for example, `object`, `OBJECT`, or `oBjEcT`), it is treated as an identifier, and a warning is issued.
- All IDL is preprocessed with the additional flag `-DIT_ORBIX3IDL_COMPATIBILITY`. This allows IDL definitions to make use of this macro in `#ifdef`s to help with migration issues.
- Unscoped types from the `CORBA` module—legacy IDL often uses `TypeCode` as a global type, whereas the IDL specification requires it to be properly scoped to the `CORBA` module. To deal with this issue, you could use the following `#ifdef` to bring `TypeCode` into global scope, if required:

```
#ifdef IT_ORBIX3IDL_COMPATIBILITY
typedef CORBA::TypeCode TypeCode;
```

```
#endif
```

Note: `TypeCode` originally was a global type in CORBA, but the `CORBA` module was added around 1992/1993 to scope such types.)

- Semicolons are tolerated in `#include` statements. The IDL compiler removes the semicolons and issues a warning.
- Opaque types—there are no easy migration solutions for opaque types. The IDL compiler does not recognize the `opaque` keyword. If you have legacy IDL that uses opaque types, you should consider migrating them to something like a `valuetype` instead.

APPENDIX 13

Mapping CORBA Exceptions

To facilitate interoperability between CORBA applications and Artix applications, Artix automatically maps between CORBA system exceptions and Artix faults.

In this appendix

This appendix discusses the following topics:

Mapping from CORBA System Exceptions

Overview

When a CORBA system exception is returned from a CORBA server to an Artix client, Artix automatically converts the CORBA system exception to a fault category.

Map from CORBA system exceptions to fault categories

Table 5 shows how each of the major CORBA system exceptions map to Artix fault categories.

Table 1: *Map from CORBA System Exceptions to Fault Categories*

CORBA System Exception	Fault Category
CORBA::BAD_CONTEXT	IT_Bus::FaultCategory::INTERNAL
CORBA::BAD_INV_ORDER	IT_Bus::FaultCategory::INTERNAL
CORBA::BAD_OPERATION	IT_Bus::FaultCategory::BAD_OPERATION
CORBA::BAD_TYPECODE	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::BAD_QOS	IT_Bus::FaultCategory::INTERNAL
CORBA::CODESET_INCOMPATIBLE	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::COMM_FAILURE	IT_Bus::FaultCategory::CONNECTION_FAILURE
CORBA::DATA_CONVERSION	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::FREE_MEM	IT_Bus::FaultCategory::MEMORY
CORBA::IMP_LIMIT	IT_Bus::FaultCategory::INTERNAL
CORBA::INITIALIZE	IT_Bus::FaultCategory::UNKNOWN
CORBA::INTERNAL	IT_Bus::FaultCategory::INTERNAL
CORBA::INTF_REPOS	IT_Bus::FaultCategory::INTERNAL
CORBA::INV_FLAG	IT_Bus::FaultCategory::INTERNAL
CORBA::INV_IDENT	IT_Bus::FaultCategory::NOT_EXIST

Table 1: *Map from CORBA System Exceptions to Fault Categories*

CORBA System Exception	Fault Category
CORBA::INV_OBJREF	IT_Bus::FaultCategory::INVALID_REFERENCE
CORBA::INV_POLICY	IT_Bus::FaultCategory::INTERNAL
CORBA::INVALID_TRANSACTION	IT_Bus::FaultCategory::INTERNAL
CORBA::MARSHAL	IT_Bus::FaultCategory::MARSHAL_ERROR
CORBA::NO_IMPLEMENT	IT_Bus::FaultCategory::NOT_IMPLEMENTED
CORBA::NO_MEMORY	IT_Bus::FaultCategory::MEMORY
CORBA::NO_PERMISSION	IT_Bus::FaultCategory::NO_PERMISSION
CORBA::NO_RESOURCES	IT_Bus::FaultCategory::INTERNAL
CORBA::NO_RESPONSE	IT_Bus::FaultCategory::INTERNAL
CORBA::OBJ_ADAPTER	IT_Bus::FaultCategory::INTERNAL
CORBA::OBJECT_NOT_EXIST	IT_Bus::FaultCategory::NOT_EXIST
CORBA::PERSIST_STORE	IT_Bus::FaultCategory::INTERNAL
CORBA::REBIND	IT_Bus::FaultCategory::INTERNAL
CORBA::TIMEOUT	IT_Bus::FaultCategory::TIMEOUT
CORBA::TRANSACTION_MODE	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSACTION_REQUIRED	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSACTION_ROLLEDBACK	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSACTION_UNAVAILABLE	IT_Bus::FaultCategory::INTERNAL
CORBA::TRANSIENT	IT_Bus::FaultCategory::TRANSIENT

Mapping from Fault Categories

Overview

When a fault (that is, a built-in exception) is returned from an Artix server to a CORBA client, Artix automatically converts the fault category to a CORBA system exception.

Map from CORBA system exceptions to fault categories

Table 6 shows how each of the Artix fault categories map to major CORBA system exceptions.

Table 2: *Map from CORBA System Exceptions to Fault Categories*

Fault Category	CORBA System Exception
IT_Bus::FaultCategory::BAD_OPERATION	CORBA::BAD_OPERATION
IT_Bus::FaultCategory::CONNECTION_FAILURE	CORBA::COMM_FAILURE
IT_Bus::FaultCategory::INTERNAL	CORBA::INTERNAL
IT_Bus::FaultCategory::INVALID_REFERENCE	CORBA::INV_OBJREF
IT_Bus::FaultCategory::LICENSE	CORBA::NO_IMPLEMENT
IT_Bus::FaultCategory::MARSHAL_ERROR	CORBA::MARSHAL
IT_Bus::FaultCategory::MEMORY	CORBA::NO_MEMORY
IT_Bus::FaultCategory::NO_PERMISSION	CORBA::NO_PERMISSION
IT_Bus::FaultCategory::NOT_EXIST	CORBA::OBJECT_NOT_EXIST
IT_Bus::FaultCategory::NOT_IMPLEMENTED	CORBA::NO_IMPLEMENT
IT_Bus::FaultCategory::NOT_UNDERSTOOD	CORBA::BAD_PARAM
IT_Bus::FaultCategory::TIMEOUT	CORBA::TIMEOUT
IT_Bus::FaultCategory::TRANSIENT	CORBA::TRANSIENT
IT_Bus::FaultCategory::UNKNOWN	CORBA::INITIALIZE
IT_Bus::FaultCategory::VERSION_ERROR	CORBA::BAD_PARAM

Mapping of Completion Status

Overview

The CORBA completion status flag and the Artix fault completion status flag have exactly the same semantics and are thus effectively equivalent. In other words, a `YES` completion status implies that the remote operation completed its work; a `NO` completion status implies that the remote operation was never called; and a `MAYBE` completion status implies that it is impossible to say whether or not the remote operation completed its work.

Completion status mapping

Table 7 shows the mapping between CORBA completion status values and fault completion status values.

Table 3: *Completion Status Mapping*

CORBA Completion Status	Fault Completion Status
`CORBA::COMPLETED_YES`	`IT_Bus::FaultCompletionStatus::YES`
`CORBA::COMPLETED_NO`	`IT_Bus::FaultCompletionStatus::NO`
`CORBA::COMPLETED_MAYBE`	`IT_Bus::FaultCompletionStatus::MAYBE`

Index