Orbix 6.3.7

Orbix Web Services

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2014-06-25

Contents

Preface

What is Covered in This Book

This book describes a variety of different CORBA integration scenarios and explains how to use the Orbix command-line tools to generate or modify WSDL contracts and IDL interfaces as required. Details of Orbix 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.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/orbix/orbix-6.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Introduction to Orbix Web Services

Orbix Web services (Orbix/WS) provides a framework for bridging between CORBA and Web Services domains. This introduction provides a brief overview of the basic integration scenarios.

Web Services and Orbix

Web services is now firmly established as one of the major frameworks for distributed systems. In particular, when it comes to exposing business applications over the Internet, many companies now prefer to use a combination of Web services with the SOAP/HTTP protocol.

When adapting an Orbix application, therefore, it frequently becomes necessary to integrate an Orbix domain with a Web services domain. Orbix/WS has been developed specifically to address this use case. Orbix/WS is a powerful integration framework that enables you to integrate Web services with Orbix, either to invoke third-party Web services from within an Orbix application or to expose an Orbix service to third-party Web service clients.

Supported transports

Orbix/WS currently supports the following protocols:

- IIOP
- SOAP/HTTP

That is, Orbix/WS is capable of performing IIOP-to-SOAP/HTTP message translation and SOAP/HTTP-to-IIOP message translation. Hence, by interposing Orbix/WS between Orbix applications and Web service applications, you can integrate the two technologies.

Mapping between IDL and WSDL

From a programmer's point of view, the key to the Web services/Orbix integration is the mapping between IDL (defining CORBA interfaces) and WSDL (defining Web service contracts). By mapping a WSDL contract to IDL, an Orbix programmer can access a Web service using the familiar tools and APIs from the world of CORBA programming.

Orbix/WS provides utilities to perform the following mappings:

- IDL-to-WSDL
- WSDL-to-IDL

Orbix router

The *Orbix router* is the runtime component of the Orbix/WS feature. Installed either as an embedded plug-in (for Orbix C++ applications) or as a standalone process (for Orbix Java applications), the Orbix router is responsible for translating messages back and forth between the IIOP protocol and the SOAP/HTTP protocol.

WSDL contract

The Web Services Definition Language (WSDL) contract plays a central role in Web services. 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 WSDL port type is analogous to an IDL interface. It defines remotely callable operations that have parameters and return values.
- *Bindings*—a WSDL 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.

  Orbix provides tools that will generate bindings for you automatically; there is no need to write them by hand.
- *Port definitions*—a WSDL port contains addressing data that enables clients to locate and connect to a remote server endpoint. For example, a CORBA port might contain stringified IOR data.

Invoking a Web Service from Orbix

This section considers the scenario where you need to integrate an Orbix application with a remote Web service. The key to this integration scenario is to map the Web services contract (WSDL file) to IDL, which enables the Orbix application to invoke the Web service as if it was a CORBA server.

Invoking a third-party Web service

Figure 1 shows the outline of this scenario, where an Orbix C++ client on host A and an Orbix Java client on host B invoke on a remote third-party Web service on host X.

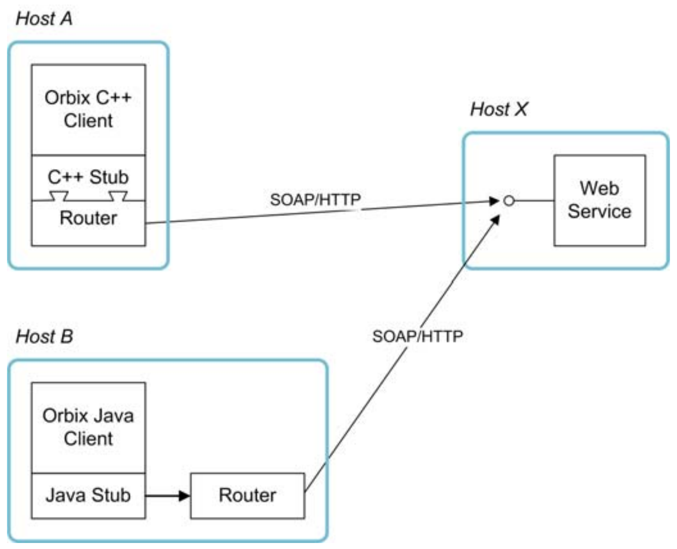

Figure 1: *Orbix Clients Invoke a Third-Party Web Service over SOAP/HTTP*

After building the Orbix application as described in this guide, you will be able to invoke the Web service from your application code using the regular C++ or Java Orbix API. From the perspective of Orbix, each Web service instance looks like a CORBA object.

C++ client architecture

In C++, the Orbix router is *embedded* in the Orbix application as a plug-in. When the C++ application code invokes an operation, the embedded Orbix router converts the invocation into a WSDL operation invocation and sends a SOAP/HTTP request to the remote Web service.

Java client architecture

In Java, the Orbix router is deployed as a *separate process* on the same host as the Orbix Java client. Because the Orbix router is implemented in C++, it is not possible to embed it into the Java client. When the Java application code invokes an operation, the stub code first of all sends an IIOP request to the Orbix router. The Orbix router then converts the invocation into a WSDL operation invocation and sends a SOAP/HTTP request to the remote Web service.

Bootstrapping the clients

Given that a Web service is represented by a CORBA object, the question then arises: how does an Orbix client get an initial reference to the CORBA object that represents the Web service? There are essentially two main bootstrap mechanisms, which are described in detail later in this guide:

- Write the IOR to a file.
- Use the CORBA Naming Service.

WSDL-to-IDL mapping

At the heart of the Web services integration is the WSDL-to-IDL mapping, which makes it possible for your Orbix clients to treat Web services as if they are CORBA objects. Figure 2 shows an overview of how the WSDL-to-IDL mapping is implemented at build time.

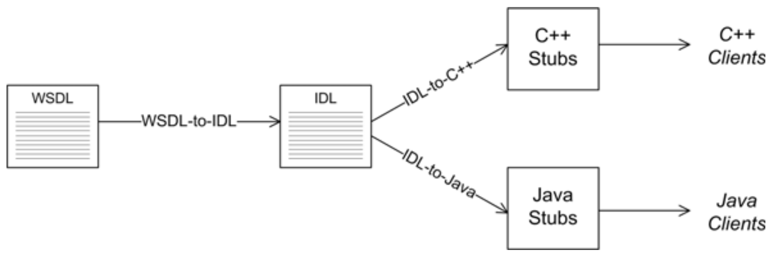

Figure 2: *Mapping WSDL to IDL*

The starting point is the WSDL file for the Web service, which you normally obtain directly from the provider of the third-party Web service. Using the tools provided by Orbix/WS, you can convert this WSDL file to an IDL file. You can then use the standard Orbix utilities to generate either C++ client stub code (for C++ clients) or Java client stub code (for Java clients).

Exposing an Orbix Service as a Web Service

This section considers the scenario where you need to expose an Orbix service as a Web service. The key to this integration scenario is to map the Orbix server's IDL interface to a Web services contract (WSDL file), thereby enabling Web services (WS) clients to access your Orbix server as if it was a Web service.

Exposing a WSDL port

Figure 3 shows the outline of this scenario, where a WS client on host A accesses a WSDL port, which is exposed by the Orbix router on behalf of a CORBA server.

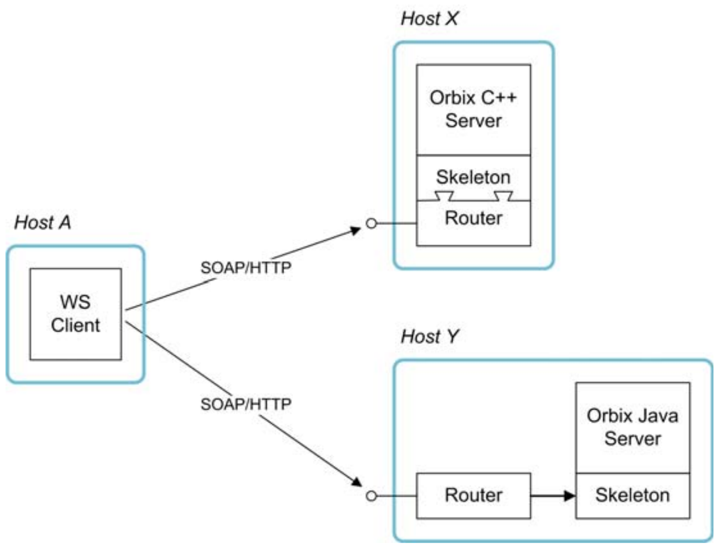

Figure 3: *WS Client Invokes Operations on a CORBA Server*

Orbix C++ server

In C++, the Orbix router is *embedded* in the Orbix application as a plug-in. When a SOAP/HTTP request is received from the remote WS client, the embedded Orbix router converts the request into a local CORBA operation invocation, which is then invoked through the C++ skeleton code.

Orbix Java server

In Java, the Orbix router is deployed as a *separate process* on the same host as the Orbix Java server. When a SOAP/HTTP request is received from the remote WS client, the Orbix router converts the request into an IIOP request, which is then invoked locally on the Orbix Java server.

Publishing a Web service

There is no analogue of the CORBA Naming Service in Web services. Unlike CORBA—which keeps the interface details (IDL interface) separate from the addressing details (IOR)—Web services keep all of the interface details and addressing details together in the WSDL file.

Hence, in order to publish a Web service, all that you need to do is to pass a copy of the WSDL file to whoever wants to use the service. If you want to manage WSDL files more systematically, however, you can use one of the many commercially available registry/repository tools for WSDL.

IDL-to-WSDL mapping

At build time, the key step that enables you to expose the Orbix service as a Web service is the IDL-to-WSDL mapping. After converting the IDL interface to WSDL, it is possible for WS clients to access an Orbix object as if it was a Web service instance. Figure 4 shows an overview of how the IDL-to-WSDL mapping is implemented at build time.

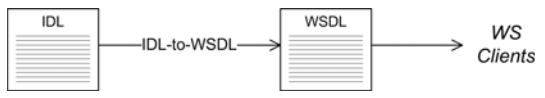

Figure 4: *Mapping IDL to WSDL*

Starting with the Orbix server's IDL file, use the tools provided by Orbix/WS to convert the IDL file to a WSDL file. You can now pass the WSDL file to any third-party WS clients that need to access the Orbix server.

CORBA Factory Pattern

One of the more advanced features of Orbix/WS is its support for the CORBA factory pattern. It turns out that this common CORBA pattern does not have a natural analogue in the Web services domain. Nevertheless, the Orbix router provides an effective and transparent solution for this use case.

Although relatively advanced, this feature is very easy to use. It is enabled by default and requires no special configuration.

Sample CORBA factory

The following IDL sample shows a typical example of the CORBA factory pattern:

```
// IDL
module WidgetDomain {
  interface Widget {
    // Various operations (not shown)
    ...
  };

  interface WidgetFactory {
    Widget getWidget();
  };
};
```

Whenever you call the `WidgetFactory::getWidget()` operation, the `WidgetFactory` creates a new `Widget` instance. Hence, the `WidgetFactory` object is capable of creating an unlimited number of `Widget` instances.

WSDL mapping of IDL interface type

When the IDL module is mapped to WSDL, all interface types are mapped to the `wsa:EndpointReferenceType` in the generated WSDL contract. The `wsa:EndpointReferenceType` type is defined by the WS-Addressing standard and in a WSDL contract it is used to represent *any* endpoint type. This convention is quite different to CORBA, where each interface is a distinct type.

Mapping the factory pattern

Now, in general, the IDL-to-WSDL mapping dictates that every CORBA object maps to a distinct Web service. This creates a peculiar problem in the case of the CORBA factory pattern, however, because a CORBA factory can create an unlimited number of CORBA objects. When these CORBA objects are mapped to the Web services domain, this implies that the CORBA factory is giving rise to *an unlimited number of Web service instances*.

The Web services framework was not originally conceived to deal with unlimited numbers of dynamically created Web services, but it turns out that the Orbix router can accommodate this feature using *dynamic proxies*.

Dynamic proxying

Dynamic proxying in the Orbix router works as follows:

1. Each time a factory operation is called (for example, `getWidget()`), the Orbix router automatically creates a dynamic proxy object for the newly-created CORBA object.
2. The dynamic proxy object is effectively a new Web service instance which wraps the new CORBA object.
3. While the Orbix router is processing the return message, the Orbix router converts the new CORBA object reference (for example, the return value of `getWidget()`) into a new `wsa:EndpointReferenceType` object that points at the new dynamic proxy object.
4. The `wsa:EndpointReferenceType` object is sent back to the WS client, which can now use it to send requests through the dynamic proxy, resulting in operation invocations on the new CORBA object.

Dynamic proxying works seamlessly and is enabled by default. The Orbix router also performs automatic garbage collection to prevent stale dynamic proxies from eating up its working memory.

Exposing an Orbix Server as a Web Service

This chapter describes how to expose an Orbix Server as a Web service using Orbix Web Services

Converting IDL to WSDL

The first step in exposing an Orbix server as a Web service is to convert the Orbix 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 IIOP 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 IIOP 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:

   ```
   > idl -wsdl <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 IDL compiler. Some of the more commonly required options are:

   `-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 IDL compiler 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 Orbix 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.

   - *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..."`

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 is set to an HTTP URL:

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

Note: It is also possible to add a SOAP endpoint to the WSDL contract using the "WSDLTOSERVICE-Transport SOAP/HTTP" command line tool.

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:wsa=http://www.w3.org/2005/08/addressing
    ...>
    ...
</definitions>
```

7. Include the WS-Addressing 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 WS-Addressing schema to represent the object references. For example, assuming that the wsaddressing.xsd schema file is stored in the same directory as router.wsdl, you can include the WS-Addressing schema in the router contract as follows:

```
<definitions name="" targetNamespace="TargetNamespaceURI"
    ...>
    <types>
        <schema targetNamespace="..." ...>
            <import
   namespace=http://www.w3.org/2005/08/addressing
   schemaLocation="wsaddressing.xsd"/>
            ...
        </schema>
    </types>
    ...
</definitions>
```

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:corba=http://schemas.iona.com/bindings/corba
xmlns:corbatm=http://schemas.iona.com/typemap/corba/cdr_over_iiop.idl
    xmlns:wsa=http://www.w3.org/2005/08/addressing
    xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
xmlns:http-conf=http://schemas.iona.com/transports/http/configuration
    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_iiop.idl
    xmlns:wsa=http://www.w3.org/2005/08/addressing
    xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
xmlns:http-conf=http://schemas.iona.com/transports/http/configuration
    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 the Orbix Router in an Orbix Server

If you are using Orbix C++, the preferred option is to embed the router in the Orbix server.

Orbix Java servers require the use of a standalone router.

Embedded Router Scenario

Figure 5 shows an overview of an Orbix router embedded in an Orbix server. In this scenario, the Orbix 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 Orbix server. Any replies from the Orbix server are then converted into SOAP/HTTP replies by the router and sent back to the client.

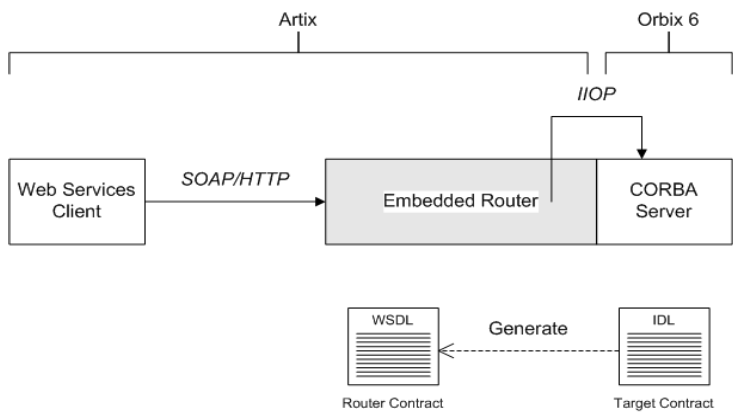

Figure 5: *Orbix Router Embedded in an Orbix C++Server*

Modifications to CORBA server

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

- Code changes - No
- Re-compilation - No
- Configuration - modify the Orbix domain configuration file or the CFR if deploying the router into an existing pre-Orbix-6.3.5 domain.

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 for the Orbix server

Embedding a Router in the Orbix Server

This section describes how to embed a router in an Orbix server. The embedded router enables the Orbix 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
- If working with an existing pre-Orbix-6.3.5 domain, edit the domain configuration.

Convert IDL to WSDL

Use the Orbix Web Services command line utilities to generate two WSDL files, `router.wsdl` and `client.wsdl`, from the Orbix server's IDL interface. For details of how to convert the IDL file to WSDL, see "Converting IDL to WSDL" on page 11.

Deploy the requisite WSDL files

Deploy the following WSDL files on the Orbix server host:

- `router.wsdl` - the router contract, which describes the route for converting SOAP/HTTP requests into Orbix requests.
- `wsaddressing.xsd` - the schema that defines the `wsa:EndpointReferenceType` data type, which Orbix uses to represent object references. The WS-Addressing schema is usually (but not always) required on the server side. If your IDL does not pass object endpoint references as parameters or return values, however, you do not need to deploy this file.

If needed, modify the Orbix domain configuration.

Note: The following is only necessary if you are working with an Orbix domain configuration deployed by Orbix version earlier than 6.3.5. Domains deployed with Orbix 6.3.5 or later contain the necessary configuration variables by default.

Given that your CORBA server is configured by a particular configuration scope, `orbix_srvr_with_embeded_router`, Example 1 shows how to modify the server configuration to embed an Orbix router.

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

```
# Orbix Configuration File


orbix_srvr_with_embedded_router {
    ...

    # Modified configuration required for embedded router:
    #
1   orb_plugins = [..., "soap", "at_http", "routing",
   "bus_loader"];;
```

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

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

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

4       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 Orbix configuration can be explained as follows:

1. Edit the ORB plug-ins, adding the requisite plug-ins to the list. In this example, the following plug-ins are needed:
   - `soap` plug-in - enables the router to send and receive SOAPmessages.
   - `at_http` plug-in - enables the router to send and receive messages over the HTTP transport.
   - `routing` plug-in - contains the core of the Orbix router.
   - `bus_loader` plug-in - triggers the Orbix router initialization step in CORBA application.
2. The Orbix 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 11). The URL can be a relative filename (as here) or a general `file://` URL
4. In order for Orbix to load the plug-ins, you must specify for each plug-in the root name of the shared library (or DLL on Windows) that contains the plug-in code.
5. In certain circumstances, Orbix creates an internal ORB instance (for example, for communication with the CFR or the Node Daemon). To prevent the settings from the current

scope being used by the internal ORBs-specifically, to prevent the internal ORB from loading the router plug-ins you should set the `share_variables_with_internal_orb` configuration variable to `false`.

Integrating the CORBA Naming Service with the Orbix Router

In a CORBA system, it is often necessary for an application to retrieve an object reference from the CORBA Naming Service. The Orbix router 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.

How the Orbix Router Resolves a Name

Figure 6 shows a typical scenario where an Orbix Router might need to resolve a name from the CORBA Naming Service. The Orbix Router, 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. However, the Naming Service is required to be deployed in the Orbix domain.

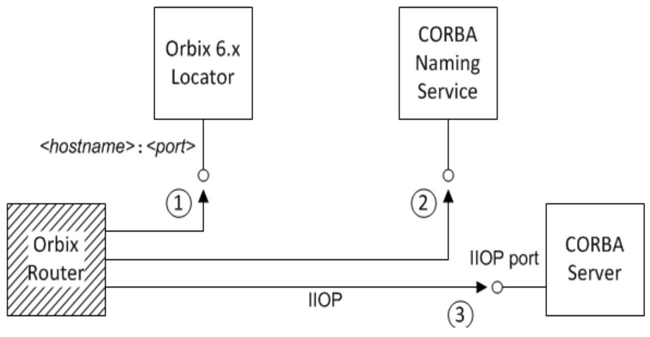

Figure 6: *Orbix Router Resolving a Name from the Naming Service*

Resolving steps for Orbix 6.x

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

Step	Action
1	The Orbix Router sends a GIOP *LocateRequest* message to the Orbix locator, whose hostname and port is specified in the Orbix domain configuration. The *LocateRequest* reply gives the location of the CORBA Naming Service.
2	The Orbix Router 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 router's WSDL contract to resolve a name from the CORBA Naming Service, you must make sure the Naming Service is deployed in the Orbix domain.

Configure the WSDL service

To configure the Orbix Router 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 `OrbixWebServicesTest` 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/OrbixWebServicesTest.obj
```

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

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

How the Orbix Router Binds a Name

Figure 7 shows a typical scenario where the Orbix Router 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 router to bind the name, you need to specify a corbaname URL in the corba:address element within a service. When the router activates the <service> or <port>, the runtime automatically binds the name in the Naming Service.

.

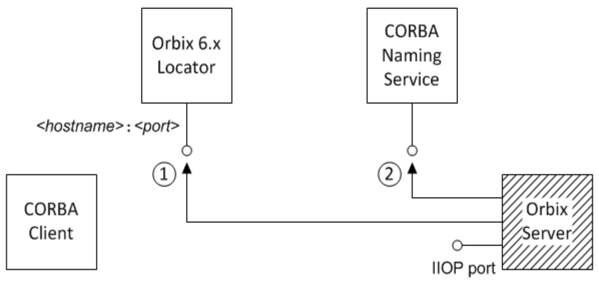

Figure 7: *Orbix Router Binding a Name to the Naming Service*

Binding steps for Orbix 6.x

Orbix Router performs the following steps to bind a name in the Orbix 6.x CORBA Naming Service (as shown in Figure 7):

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

Prerequisites

Before configuring the router's WSDL contract to resolve a name from the CORBA Naming Service, you must make sure the Naming Service is deployed in the Orbix domain.

Configure the WSDL service

To configure an Orbix Router 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 resolving the name, but the router treats this configuration setting differently. When the router activates a service containing a `corbaname` URL, it automatically binds the given *StringName* into the CORBA Naming Service.

Binding semantics

The automatic binding performed by the router 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 router 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 router will not bind the given name. You would need to create the naming contexts manually (for example, you could issue the command `itadmin ns newnc` *NameContext*).

Advanced CORBA Port Configuration

This chapter describes some advanced configuration options for customizing a CORBA port in the Orbix Router WSDL

Configuring Fixed Ports and Long-Lived IORs

The Orbix Router 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 router runs.
- *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`.

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 Orbix Router runs.

- *Long-lived IORs* - the CORBA binding generates IORs in such a way that they remain valid even when the router 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 CORBA Web Service to be direct persistent

To configure a router CORBA service to be direct persistent, you must edit both the WSDL file and the Orbix configuration.

Editing the WSDL file

The Orbix Router enables you to set direct persistence attributes in WSDL by adding a `corba:policy` element to the WSDL service, as shown in Example 2. The `corba:policy` attributes from Example 3 can be explained as follows:.

Example 2: *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 2 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 Orbix Router 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. Currently, the Orbix Router 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.

  Currently, the Orbix Router currently, allows you to set only one Object ID for each port.

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

Editing Orbix configuration

To complete the configuration of direct persistence, you must also set some configuration variables in the relevant scope of the Orbix configuration.

For example, if your Orbix Router uses the orbix_router configuration scope, you would add the configuration variables as shown in Example 3.

Example 3: *Setting Direct Persistence Configuration Variables*

```
# Orbix configuration
...
orbix_router {
   ...
   poa:FQPN:direct_persistent="true";
   poa:FQPN:well_known_address="WKA_prefix";
   WKA_prefix:iiop:port="IP_Port";
};
```

The configuration variables from Example 3 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 2 on page 28).

  Note: In CORBA terms, this is equivalent to setting the `IT_PortableServer::PersistenceModePolicy` policy to `DIRECT_PERSISTENCE`.

- `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 messages. This port value is also published in the IIOP profile of generated IORs.

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

Restricts the IIOP 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 interface 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 Orbix Security Guide for details of how to configure IIOP/TLS security.

CORBA Timeout Policies

The Orbix Router that exposes 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 Orbix Router configuration scope in Orbix configuration. For example, for an Orbix Router that uses the `orbix_router` configuration scope, you can set the CORBA relative round trip timeout as follows:

```
# Orbix Configuration
orbix_router{
    # 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 Orbix configuration:

`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

Orbix 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.

Orbix IDL-to-WSDL Mapping

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

Introducing CORBA Type Mapping

To ensure that messages are converted into the proper format for a CORBA application to understand, WSDL contracts need to unambiguously describe how data is mapped to CORBA data types. For primitive types, the mapping is straightforward. However, complex types such as structures, arrays, and exceptions require more detailed descriptions.

Unsupported types

The following CORBA types are not supported:

- Value types
- Boxed values
- Local interfaces
- Abstract interfaces
- Forward-declared interfaces
- Preprocessor include directives

Preprocessor include directives

When converting IDL to WSDL, you can use either of the following preprocessor include directives in your IDL code:

```
#include "IncludedFile"
#include <IncludedFile>
```

Both of these include directives are processed in the same way: the preprocessor searches for the specified IDL files in the current include path. The include path consists of the directories specified using the -I option on the IDL-to-WSDL compiler command line (idl -wsdl). For example:

```
idl -wsdl -I FirstIncludeDir -I SecondIncludeDir ...
```

IDL Primitive Type Mapping

Mapping chart

Most primitive IDL types are directly mapped to primitive XML Schema types. Table 1 lists the mappings for the supported IDL primitive types.

Table 1: *Primitive Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type
`any`	`xsd:anyType`	`corba:any`
`boolean`	`xsd:boolean`	`corba:boolean`
`char`	`xsd:byte`	`corba:char`
`string`	`xsd:string`	`corba:string`
`wchar`	`xsd:string`	`corba:wchar`
`wstring`	`xsd:string`	`corba:wstring`
`short`	`xsd:short`	`corba:short`
`long`	`xsd:int`	`corba:long`
`long long`	`xsd:long`	`corba:longlong`
`unsigned short`	`xsd:unsignedShort`	`corba:ushort`
`unsigned long`	`xsd:unsignedInt`	`corba:ulong`
`unsigned long long`	`xsd:unsignedLong`	`corba:ulonglong`
`float`	`xsd:float`	`corba:float`
`double`	`xsd:double`	`corba:double`
`long double`	*Not Supported*	*Not Supported*
`octet`	`xsd:unsignedByte`	`corba:octet`
`fixed`	`xsd:decimal`	`corba:fixed`
`Object`	`wsa:EndpointReferenceType`	`corba:object`
`TimeBase::UtcT`	`xsd:dateTime`[a]	`corba:dateTime`

a. The mapping between `xsd:dateTime` and `TimeBase:UtcT` is only partial. For the restrictions see "Unsupported time/date values" on page 37

Unsupported types

The Orbix Router does not support the CORBA long double type.

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.

String type

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`).

Fixed type

The mapping of fixed is a special case. Although fixed maps directly to the `xsd:decimal` type, Orbix must store additional mapping information to support round-trip conversion between WSDL and IDL. Therefore, Orbix records the details of the IDL fixed mapping in a `corba:fixed` element (within the scope of the `corba:typeMapping` element). For example, the mapping of a `fixed<6,2>` type might be recorded as follows:

```
<corba:typeMapping ... >
   <corba:fixed digits="6"
                scale="2"
                name="SampleTypes.Money"
                repositoryID="IDL:SampleTypes/Money:1.0"
                type="xsd:decimal"/>
</corba:typeMapping>
```

Example

The mapping of primitive types is handled in the CORBA binding section of the WSDL contract. For example, consider an input message that has a part, score, that is described as an `xsd:int` as shown in Example 4.

Example 4: *WSDL Operation Definition*

```
<message name="runsScored">
    <part name="score"/>
</message>
<portType ...>
    <operation name="getRuns">
        <input message="tns:runsScored" name="runsScored"/>
    </operation>
</portType>
```

It is described in the CORBA binding as shown in Example 5.

Example 5: *Example CORBA Binding*

```
<binding ...>
  <operation name="getRuns">
    <corba:operation name="getRuns">
     <corba:param name="score" mode="in" idltype="corba:long"/>
    </corba:operation>
      <input/>
      <output/>
  </operation>
</binding>
```

The IDL is shown in Example 6.

Example 6: *getRuns IDL*

```
// IDL
void getRuns(in score);
```

IDL Complex Type Mapping

This section describes how the complex IDL data types are mapped to WSDL. It contains the following types:

- IDL enum Type
- IDL struct Type
- IDL union Type
- IDL sequence Types
- IDL array Types
- IDL exception Types
- IDL typedef Expressions

IDL enum Type

An IDL enumeration maps to an XML string with enumeration facets. The mapped enumeration is a simple type derived by restriction from the `xsd:string` type.

IDL example

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

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

WSDL mapping

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>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Orbix records the details of the enumeration type mapping in a `corba:enum` element (within the scope of the `corba:typeMappin` element), as follows:

```
corba:typeMapping ... >
  <corba:enum name="SampleTypes.Shape"
    repositoryID="IDL:SampleTypes/Shape:1.0"
    type="xsd1:SampleTypes.Shape">
    <corba:enumerator value="Square"/>
    <corba:enumerator value="Circle"/>
    <corba:enumerator value="Triangle"/>
   </corba:enum>
...
</corba:typeMapping>
```

IDL struct Type

An IDL structure maps to an `xsd:sequence` type. Each field in the IDL structure maps to an element in the sequence.

IDL example

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

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

WSDL mapping

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>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Orbix records the details of the structure type mapping in a `corba:struct` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
     <corba:struct name="SampleTypes.SampleStruct"
   repositoryID="IDL:SampleTypes/SampleStruct:1.0"
   type="xsd1:SampleTypes.SampleStruct">
        <corba:member idltype="corba:string" name="theString"/>
         <corba:member idltype="corba:long" name="theLong"/>
    </corba:struct>
</corba:typeMapping>
```

IDL union Type

Unions are particularly difficult to describe using the XML schema framework. In the logical data type descriptions, the difficulty is how to describe the union without losing the relationship between the members of the union and the discriminator used to select the members. The easiest method is to describe a union using an `xsd:choice` and list the members in the specified order. The OMG's proposed method is to describe the union as an `xsd:sequence` containing one element for the discriminator and an `xsd:choice` to describe the members of the union. However, neither of these methods can accurately describe all the possible permutations of a CORBA union.

IDL example

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

```
// IDL
module SampleTypes {
    union Poly switch (short)
    {
      case 0:
        string StringCase0;
      case 1:
      case 2:
        float FloatCase1and2;
      default:
        long caseDef;
    };
};
```

WSDL mapping—default

The IDL-to-WSDL compiler generates the following mapping for the IDL union type by default:

```
<complexType name="SampleTypes.Poly">
    <choice>
        <element name="StringCase0" type="string"/>
        <element name="FloatCase1and2" type="float"/>
        <element name="caseDef" type="int"/>
    </choice>
</complexType>
```

In this case, the IDL union maps to `xsd:choice`, where the name of the type is `SampleTypes.Poly`. By default, Orbix uses the `xsd:choice type` as the representation of the union throughout the contract.

WSDL mapping—OMG alternative

The IDL-to-WSDL compiler also generates the following alternative mapping for the IDL union type:

```
<complexType name="SampleTypes._omg_Poly">
    <sequence>
        <element maxOccurs="1" minOccurs="1"
   name="discriminator"
                        type="short"/>
        <choice maxOccurs="1" minOccurs="0">
            <element name="StringCase0" type="string"/>
            <element name="FloatCase1and2" type="float"/>
            <element name="caseDef" type="int"/>
        </choice>
    </sequence>
</complexType>
```

In this case, the IDL union maps to `xsd:sequence`, where the name of the type is obtained by prepending **`_omg_`** to the basic type name, giving `SampleTypes._omg_Poly`.

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Orbix records the details of the union type mapping in a `corba:union` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:union discriminator="corba:short" name="SampleTypes.Poly"
   repositoryID="IDL:SampleTypes/Poly:1.0"
   type="xsd1:SampleTypes.Poly">
        <corba:unionbranch idltype="corba:string" name="StringCase0">
            <corba:case label="0"/>
        </corba:unionbranch>
        <corba:unionbranch idltype="corba:float" name="FloatCase1and2">
            <corba:case label="1"/>
            <corba:case label="2"/>
        </corba:unionbranch>
        <corba:unionbranch default="true" idltype="corba:long"
   name="caseDef"/>
    </corba:union>
</corba:typeMapping>
```

IDL sequence Types

An IDL sequence maps to a sequence containing a single element that has `minOccurs` equal to zero and `maxOccurs` equal to the sequence's upper bound (`maxOccurs` equals unbounded, for an unbounded sequence).

IDL example

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

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

WSDL mapping

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>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Orbix records the details of the IDL sequence type mapping in a `corba:sequence` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:sequence bound="0"

   elemtype="corbatm:SampleTypes.SampleStruct"
                    name="SampleTypes.SeqOfStruct"

   repositoryID="IDL:SampleTypes/SeqOfStruct:1.0"
                    type="xsd1:SampleTypes.SeqOfStruct"/>
</corba:typeMapping>
```

IDL array Types

An IDL array maps to a sequence containing a single element that sets both minOccurs and maxOccurs equal to the array bound.

IDL example

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

```
// IDL
module SampleTypes {
     typedefSampleStructArrOfStruct[10];
    ...
};
```

WSDL mapping

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>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Orbix records the details of the IDL array type mapping in a corba:array element (within the scope of the corba:typeMapping element), as follows:

```
<corba:typeMapping ... >
    <corba:array bound="10"

   elemtype="corbatm:SampleTypes.SampleStruct"
                 name="SampleTypes.ArrOfStruct"

   repositoryID="IDL:SampleTypes/ArrOfStruct:1.0"
                 type="xsd1:SampleTypes.ArrOfStruct"/>
</corba:typeMapping>
```

IDL exception Types

An IDL exception type maps to an `xsd:sequence` type and to an exception message. Each field in the IDL exception maps to an element in the `xsd:sequence`.

IDL example

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

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

WSDL mapping

The IDL-to-WSDL compiler maps the `SampleTypes::GenericExc` exception to a WSDL sequence type, `SampleTypes.GenericExc`c, and to a WSDL fault message, `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="SampleTypes.GenericExc">
        <part element="xsd1:SampleTypes.GenericExc"
   name="exception"/>
</message>
```

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Orbix records the details of the IDL exception type mapping in a `corba:exception` element (within the scope of the `corba:typeMapping` element).

The IDL-to-WSDL compiler generates the following `corba:exception` element:

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

<corba:typeMapping ... >
     <corba:exception name="SampleTypes.GenericExc"
           repositoryID="IDL:SampleTypes/GenericExc:1.0"
            type="xsd1:SampleTypes.GenericExc">
        <corba:member idltype="corba:string" name="reason"/>
     </corba:exception>
</corba:typeMapping>
```

IDL typedef Expressions

If a type is aliased in IDL, using a typedef expression, Orbix simply replaces the type alias with the original type when mapping to WSDL.

IDL example

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

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

CORBA type mapping

To support round-trip conversion between WSDL and IDL, Orbix records the details of each IDL alias mapping in a `corba:alias` element (within the scope of the `corba:typeMapping` element), as follows:

```
<corba:typeMapping ... >
    <corba:alias basetype="corba:float"
              name="SampleTypes.FloatAlias"

   repositoryID="IDL:SampleTypes/FloatAlias:1.0"
              type="xsd:float"/>
    <corba:alias
   basetype="corbatm:SampleTypes.SampleStruct"
              name="SampleTypes.SampleStructAlias"

   repositoryID="IDL:SampleTypes/SampleStructAlias:1.0"
              type="xsd1:SampleTypes.SampleStruct"/>
</corba:typeMapping>
```

WSDL mapping

The IDL-to-WSDL compiler maps the `SampleTypes::FloatAlias` type alias directly to the type, `xsd:float` and the `SampleTypes::SampleStructAlias` type alias directly to the type, `SampleTypes.SampleStruct`.

Configuring a CORBA Binding

CORBA bindings are described using a variety of Orbix-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 conventionally prefixed by the namespace prefix, `corba`.

The following is the definition of the corba namespace prefix:

```
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 Orbix-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 7.

Example 7: *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 8.

Example 8: *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 Orbix-specific WSDL elements, corba:address and corba:policy, within the WSDL port element, to specify how a CORBA object is exposed.

Namespace

- The WSDL extensions used to describe CORBA data mappings and CORBA transport details are conventionally prefixed by the namespace prefix, `corba`. The definition of the corba namespace prefix is

```
xmlns:corba="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 WSDL 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 Orbix process is started in, not relative to the containing WSDL file.

- Specify that the IOR is published to the Naming Service, by entering the object's name using the `corbaname` format:

```
corbaname:rir/NameService#object_name
```

  For more information on using the Naming Service with Orbix Web Services see Integrating the CORBA Naming Service with the Orbix Router page 23
- 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 Orbix Router 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 Micro Focus's Orbix, such as load balancing and fault tolerance. For information on using these advanced CORBA features, see the Orbix Documentation Library.

POA Name

By default, a router 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 Orbix 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 Orbix 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 9:

Example 9: *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>
```

Orbix expects the IOR for the CORBA object to be located in a file called `objref.ior` (relative to the directory in which the Orbix process is started), and creates a persistent POA with an object id of `personalInfo` to connect the CORBA application.

Web Services Utilities in Orbix

Use the Orbix Web Services command-line utilities to convert OMG IDL to WSDL and to generate CORBA bindings.

Converting OMG IDL to WSDL

Micro Focus'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.

Location

The location of the utility tool is *`$IT_PRODUCT_DIR/asp/6.3/bin`*.

WSDLTOCORBA / WSDLTOIDL

Synopsis

```
idl -wsdl [-I idl-include-dir...] [-o output-dir] [-a
corba-address] [-b] [-f corba-address-file] [-n
   schema-import-file] [-s idl-sequence-type] [-w
   target-namespace] [-x schema-namespace]
[-t corba-typemap-namespace] [-L logical-wsdl-filename] [-P
physical-wsdl-filename] [-T schema-filename] [-qualified] [-e
xml-encoding-type] [-mnsnamespaceMapping] [-ow wsdloutput-file]
[exexcludedModules] [-pf] [-v] [[-verbose] | [-quiet]] idl
```

Required Arguments

The command has the following required arguments:

`idl`	Specifies the name of the IDL file.

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. You can use this flag multiple times.
`-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 file must exist when you run the IDL compiler.
`-n` *`schema-import-file`*	Specifies that a schema 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 document's target namespace.
`-x` *`schema-namespace`*	Specifies the namespace to use for the generated XML Schema's target namespace.
`-t` *`corba-typemap-namespace`*	Specifies the namespace to use for the CORBA type map's target namespace.
`-L` *`logical-wsdl-filename`*	Specifies that the logical portion of the generated WSDL specification into is written to `logical-wsdl-filename`. The logical WSDL is then imported into the default generated file.
`-P` *`physical-wsdl-filename`*	Specifies that the physical portion of the generated WSDL specification into is written to `physical-wsdl-filename`. The physical WSDL is then imported into the default generated file.
`-T` *`schema-filename`*	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.
`-qualified`	Generates fully qualified WSDL.
`-e` *`xml-encoding-type`*	Specifies the value for the generated WSDL document's xml encoding attribute. the generated WSDL document's xml encoding attribute. The default is UTF-8.
`-mnsnamespaceMapping`	Specifies a mapping between IDL modules and XML namespaces.
`-ow` *`wsdloutput-file`*	Specifies the name of the generated WSDL file.

`-exexcludeModules`	Specifies one or more IDL modules to exclude when generating the WSDL file.
`-pf`	Specifies that polymorphic factory support is enabled.
`-h`	Displays the tool's usage statement.
`-v`	Displays the version number for the tool.
`-verbose`	Displays comments during the code generation process.
`-quiet`	Suppresses comments during the code generation process.

Generating a Deployment Descriptor

WSDD generates a deployment descriptor that can be used to deploy the Orbix Router into the Orbix Router container.

Location

The location of the utility tool is *$IT_PRODUCT_DIR/asp/6.3/bin*.

Deploy the Orbix Router into the Orbix Router container

Synopsis

```
wsdd {-service QName} {-pluginName name} {-pluginType { Cxx
   | Java }}[-pluginImpl name] [-pluginURL dir] [-wsdlurl
   URL] [-provider namespace][-file file] [-d dir]
   [[-quiet] | [-verbose]] [-h] [-v]
```

Required Arguments

The command has the following required arguments:

`-service` *QName*	Specifies the QName of the plug-in’s service as given in its contract.
`-pluginName` *name*	Specifies the name of the plug-in as specified in the Orbix configuration.
`-pluginType{Cxx}`	Specifies that the plug-in is implemented as a shared library.

Options

The command has the following options:

`-pluginImpl` *name*	Specifies the library name of the plug-in’s implementation.
`-pluginURL` *dir*	Specifies the directory where the plug-in’s implementation is located.
`-wsdlurl` *URL*	Specifies the location of the contract defining the service implemented by the plug-in.

`-provider` *`namespace`*	Specifies the namespace under which your plug-in's `ServantProvider` is registered with the bus.
`-file` *`file`*	Specifies the name of the generated deployment descriptor.
`-d` *`dir`*	Specifies the directory where the generated file will be written.
`-quiet`	Specifies that the tool is to run in quiet mode.
`-verbose`	Specifies that the tool is to run in verbose mode.
`-h`	Displays the tool's usage statement.
`-v`	Displays the tool's version.

Generating a CORBA Binding

Adds a CORBA binding to a WSDL document. The generated WSDL file will also contain a CORBA port with no address specified.

Location

The location of the utility tool is *`$IT_PRODUCT_DIR/asp/6.3/bin`*.

WSDLTOCORBA -CORBA

Synopsis

```
wsdltocorba -corba {-i portType} [-d dir] [-b binding] [-o file][-props
   namespace] [-wrapped] [-L file] [[-quiet] | [-verbose]] [-h] [-v]
   wsdl
```

Required Arguments

The command has the following required arguments:

`-i` *`portType`*	Specifies the name of the port type for which the CORBA binding is generated.
`wsdl`	Specifies the WSDL document to which the binding is added.

Options

The command has the following options:

`-d` *`dir`*	Specifies the directory into which the new WSDL document is written.
`-b` *`binding`*	Specifies the name of the generated CORBA binding. The default is `portTypeBinding`.
`-o file`	Specifies the name of the generated WSDL document. The default is `wsdl_file-corba.wsdl`.

`-props namespace`	Specifies the namespace to use for the generated CORBA typemap.
`-wrapped`	Specifies that the generated binding uses wrapped types.
`-L file`	Specifies the location of your Orbix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-quiet`	Specifies that the tool is to run in quiet mode.
`-verbose`	Specifies that the tool is to run in verbose mode.
`-h`	Displays the tool's usage statement.
`-v`	Displays the tool's version.

Adding a Route

Adds a route to a WSDL document. Routes are used by the Orbix router to direct messages between endpoints.

Location

The location of the utility tool is *$IT_PRODUCT_DIR/asp/6.3/bin*.

WSDLROUTING

Synopsis

```
wsdltorouting [-rn name ] [-ssn service] [-spn port] [-dsn service][-dpn
   port] [-on operation] [-ta attribute] [-d dir] [-o file] [-L
   file][[-quiet] | [-verbose]] [-h] [-v] {wsdl}
```

Options

The command has the following options:

`-rn name`	Specifies the name of the generated route. If no name is given a unique name will be generated for the route.
`-ssn service`	Specifies the name of the service to use as the source of the route.
`-spn port`	Specifies the name of the port to use as the source of the route. The port must correspond to a `port` element in the specified service.
`-dsn service`	Specifies the name of the service to use as the destination of the route.
`-dpn port`	Specifies the name of the port to use as the destination of the route. The port must correspond to a `port` element in the specified service.
`-on operation`	Specifies the name of the operation to use for the route. If the route is port-based, you do need to use this flag.

`-ta` *attribute*	Specifies a transport to use in defining the route.
`-d` *dir*	Specifies the output directory for the generated contract.
`-o` *file*	Specifies the filename for the generated contract.
`-L` *file*	Specifies the location of your Orbix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-quiet`	Specifies that the tool is to run in quiet mode.
`-verbose`	Specifies that the tool is to run in verbose mode.
`-h`	Displays the tool's usage statement.
`-v`	Displays the tool's version.
wsdl	Specifies the name of the WSDL document to which the route is added.

Generating an HTTP Endpoint

Generates a WSDL document containing an HTTP endpoint.

Location

The location of the utility tool is *$IT_PRODUCT_DIR/asp/6.3/bin*.

WSDLTOSERVICE-Transport SOAP/HTTP

Synopsis

```
wsdltoservice -transport soap/http [-e service] [-t port] [-b binding]
    [-a address] [-hssdt serverSendTimeout] [-hscvt serverReceiveTimeout]
    [-hstrc trustedRootCertificates] [-hsuss useSecureSockets] [-hsct
    contentType] [-hscc serverCacheControl] [-hsscse
    supressClientSendErrors] [-hsscre supressClientReceiveErrors] [-hshka
    honorKeepAlive] [-hsrurl redirectURL] [-hscl contentLocation] [-hsce
    contentEncoding] [-hsst serverType] [-hssc serverCentificate] [-hsscc
    serverCentificateChain] [-hsspk serverPrivateKey] [-hsspkp
    serverPrivateKeyPassword] [-hcst clientSendTimeout] [-hccvt
    clientReceiveTimeout] [-hctr trustedRootCertificates] [-hcuss
    useSecureSockets] [-hcct contentType] [-hccc clientCacheControl]
    [-hcar autoRedirect] [-hcun userName] [-hcp password] [-hcat
    clientAuthorizationType] [-hca clientAuthorization] [-hca accept]
    [-hcal acceptLanguage] [-hcae acceptEncoding] [-hch host] [-hccn
    clientConnection] [-hcck cookie] [-hcbt browserType] [-hcr referer]
    [-hcps proxyServer] [-hcpun proxyUserName] [-hcpp proxyPassword]
    [-hcpat proxyAuthorizationType] [-hcpa proxyAuthorization] [-hccce
    ClientCertificate] [-hcccc clientCertificateChain] [-hcpk
    clientPrivateKey] [-hcpkp clientPrivateKeyPassword] [-o file] [-d
    dir] [-L file] [[-quiet] | [-verbose]] [-h] [-v] wsdlurl
```

Required Arguments

The command has the following required arguments:

wsdlurl	Specifies the WSDL document from which to base the generated WSDL document.

Options

The command has the following options:

`-transport soap/http`	If the payload being sent over the wire is SOAP, use `-transport soap`. For all other payloads use `-transport http`.
`-e` *service*	Specifies the name of the generated service.
`-t` *port*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *binding*	Specifies the name of the binding for which the service is generated.
`-a` *address*	Specifies the value used in the `address` element of the port.
`-hssdt` *serverSendTimeout*	Specifies the number if milliseconds that the server can continue to try to send a response to the client before the connection is timed out.
`-hscvt` *serverReceiveTimeout*	Specifies the number of milliseconds that the server can continue to try to receive a request from the client before the connection is timed out.
`-hstrc` *trustedRootCertificates*	Specifies the full path to the X509 certificate for the certificate authority.
`-hsuss` *useSecureSockets*	Specifies if the server uses secure sockets. Valid values are true or false.
`-hsct` *contentType*	Specifies the media type of the information being sent in a server response.
`-hscc` *serverCacheControl*	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
`-hsscse` *supressClientSendErrors*	Specifies whether exceptions are thrown when an error is encountered on receiving a client request. Valid values are true or false.
`-hsscre` *supressClientReceiveErrors*	Specifies whether exceptions are thrown when an error is encountered on sending a response to a client. Valid values are true or false.
`-hshka` *honorKeepAlive*	Specifies if the server honors client keep-alive requests. Valid values are true or false.

`-hsrurl` *`redirectURL`*	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource.
`-hscl` *`contentLocation`*	Specifies the URL where the resource being sent in a server response is located.
`-hsce` *`contentEncoding`*	Specifies what additional content codings have been applied to the information being sent by the server, and what decoding mechanisms the client therefore needs to retrieve the information.
`-hsst` *`serverType`*	Specifies what type of server is sending the response to the client.
`-hssc` *`serverCentificate`*	Specifies the full path to the X509 certificate issued by the authority for the server.
`-hsscc` *`serverCentificateChain`*	Specifies the full path to the file that contains all the certificates in the chain.
`-hsspk` *`serverPrivateKey`*	Specifies the full path to the private key that corresponds to the X509 certificate specified by `serverCertificate`.
`-hsspkp` *`serverPrivateKeyPassword`*	Specifies a password that is used to decrypt the private key.
`-hcst` *`clientSendTimeout`*	Specifies the number of milliseconds that the client can continue to try to send a request to the server before the connection is timed out.
`-hccvt` *`clientReceiveTimeout`*	Specifies the number of milliseconds that the client can continue to try to receive a response from the server before the connection is timed out.
`-hctrc` *`trustedRootCertificates`*	Specifies the full path to the X509 certificate for the certificate authority.
`-hcuss` *`useSecureSockets`*	Specifies if the client uses secure sockets. Valid values are `true` or `false`.
`-hcct` *`contentType`*	Specifies the media type of the data being sent in the body of the client request.
`-hccc` *`clientCacheControl`*	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
`-hcar` *`autoRedirect`*	Specifies if the server should automatically redirect client requests.

-hcun *userName*	Specifies the username the client uses to register with servers.
-hcp *password*	Specifies the password the client uses to register with servers.
-hcat *clientAuthorizationType*	Specifies the authorization mechanisms the client uses when contacting servers.
-hca *clientAuthorization*	Specifies the authorization credentials used to perform the authorization.
-hca *accept*	Specifies what media types the client is prepared to handle.
-hcal *acceptLanguage*	Specifies what language the client prefers for the purposes of receiving a response.
hcae *acceptEncoding*	Specifies what content codings the client is prepared to handle.
-hch *host*	Specifies the internet host and port number of the resource on which the client request is being invoked.
-hccn *clientConnection*	Specifies if the client will open a new connection for each request or if it will keep the original one open. Valid values are close and Keep-Alive.
-hcck *cookie*	Specifies a static cookie to be sent to the server.
-hcbt *browserType*	Specifies information about the browser from which the client request originates.
-hcr *referer*	Specifies the value for the client’s referring entity.
-hcps *proxyServer*	Specifies the URL of the proxy server, if one exists along the message path.
-hcpun *proxyUserName*	Specifies the username that the client uses to authorize with proxy servers.
-hcpp *proxyPassword*	Specifies the password that the client uses to authorize with proxy servers.
-hcpat *proxyAuthorizationType*	Specifies the authorization mechanism the client uses with proxy servers.
-hcpa *proxyAuthorization*	Specifies the actual data that the proxy server should use to authenticate the client.
-hccce *ClientCertificate*	Specifies the full path to the X509 certificate issued by the certificate authority for the client.
-hcccc *clientCertificateChain*	Specifies the full path to the file that contains all the certificates in the chain.

`-hcpk` *`clientPrivateKey`*	Specifies the full path to the private key that corresponds to the X509 certificate specified by `clientCertificate`.
`-hcpkp` *`clientPrivateKeyPassword`*	Specifies a password that is used to decrypt the private key.
`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d` *`dir`*	Specifies the output directory for the generated contract.
`-L` *`file`*	Specifies the location of your Orbix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-quiet`	Specifies that the tool runs in quiet mode.
`-verbose`	Specifies that the tool runs in verbose mode.
`-h`	Displays the tool’s usage statement.
`-v`	Displays the tool’s version.

Generating a SOAP Binding

Generates a WSDL document containing a SOAP binding to a WSDL document based on the values provided as arguments to the tool.

Location

The location of the utility tool is *`$IT_PRODUCT_DIR/asp/6.3/bin`*.

WSDLTOSOAP

Synopsis

```
wsdltosoap {-i portType} {-n namespace} [-soapversion [ 1.1 | 1.2
   ]][-style [ document | rpc ]] [-use [ literal | encoded ]] [-b
   binding] [-o file][-d dir] [-L file] [[-quiet] | [-verbose]] [-h]
   [-v] wsdlurl
```

Required Arguments

The command has the following required arguments:

`-i` *`portType`*	Specifies the name of the `portType` element being mapped to a SOAP binding.
`-n` *`namespace`*	Specifies the namespace to use for the SOAP binding.

wsdlurl	Specifies the WSDL document from which to base the generated WSDL document.

The command has the following options:

-soapversion [1.1 | 1.2]	Specifies the SOAP version of the generated binding. Defaults to 1.1.
-style [document | rpc]	Specifies the encoding style to use in the SOAP binding. Defaults to document.
-use [literal | encoded]	Specifies how the data is encoded. Default is literal.
-o *file*	Specifies the filename for the generated contract. The default is to append -service to the name of the imported contract.
-d *dir*	Specifies the output directory for the generated contract.
-L file	Specifies the location of your Orbix license file. The default behavior is to check IT_PRODUCT_DIR\etc\license.txt.
-quiet	Specifies that the tool runs in quiet mode.
-verbose	Specifies that the tool runs in verbose mode.
-h	Displays the tool's usage statement.
-v	Displays the tool's version.

Index