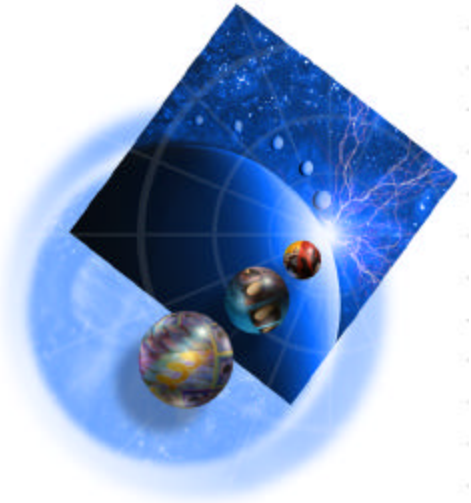
Whitepaper

IBM WebSphere

IBM WebSphere Application Servers

CORBA Interoperability



IBM WebSphere
Document Version 4

Table Of Contents

Samples 41

Notices

Trademarks and Service Marks

Preface

This document describes WebSphere CORBA interoperability for the version 5 IBM® WebSphere® Application products. These products are:

- IBM WebSphere Application Server Version 5
- IBM WebSphere Application Server Enterprise Version 5
- IBM WebSphere Application Server Version 5 for z/OS

For a description of WebSphere CORBA interoperability for previous versions of IBM WebSphere Application Server products, refer to *WebSphere Application Servers CORBA Interoperability, document version 2.3*.

Terms

This document uses the generic term *version 5 IBM WebSphere Application Server products* to refer to any of the version 5 IBM WebSphere Application Server products. This document uses the name of an IBM WebSphere Application Server product to refer to the specific product.

Audience

This document is intended for customers of version 5 IBM WebSphere Application Server products who have deployed or plan to deploy solutions that require interoperation with CORBA clients, CORBA servers, or both.

Organization

This document is organized as follows:

- Section 1 provides an introduction to WebSphere CORBA interoperability.
- Section 2 discusses the similarities and differences between the WebSphere EJB and CORBA programming models.
- Section 3 describes how to configure WebSphere CORBA interoperability.
- Section 4 discusses some interoperability problems that might occur and suggests workarounds for these problems.
- Section 5 describes samples that demonstrate and test WebSphere CORBA interoperability.

Summary of Major Changes

The following is a summary of major changes affecting WebSphere CORBA interoperability between version 5 and prior versions of the WebSphere Application Server products:

- In version 5, full support has been added for the CORBA Interoperability Name Service (INS) specification.
- In version 5, full support has been added for version 1.2 of the General Inter-ORB (GIOP) protocol.
- In version 5, co-existence is no longer supported.

Related Documents

Java™ Developer Connection, Writing Advanced Applications, CORBA
`http://developer.java.sun.com/developer/onlineTraining/Programming/ JDCBook/corba.html`

The Common Object Request Broker: Architecture and Specification, version 2.3 (earlier versions are also helpful)
`ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf`

Java Language to IDL Mapping Specification
`ftp://ftp.omg.org/pub/docs/formal/99-07-59.pdf`

IDL to Java Language Mapping Specification
`ftp://ftp.omg.org/pub/docs/formal/99-07-59.pdf`

C++ Language Mapping Specification
`ftp://ftp.omg.org/pub/docs/formal/99-07-41.pdf`

Disclaimer

This document provides general technical information concerning the ability of IBM WebSphere Object Request Brokers (“ORBs”), and non-IBM ORBs to operate together. Any sample software code that may be provided in conjunction with this document provides more detailed technical information concerning the ability of IBM ORBs and certain non-IBM ORBs to operate together within a specific environment, and is provided for internal, demonstration purposes relating to the interoperability of IBM and non-IBM ORBs only.

IBM may, at its sole discretion, provide technical support services with respect to the operation of IBM ORBs in conjunction with non-IBM ORBs, or with respect to any sample software code provided herewith. IBM reserves the right to limit the scope

of such services in any manner it chooses, to refuse to provide such services, or to terminate such services at any time, without notice.

The level of interoperability experienced between IBM ORBs and another non-IBM ORB depends upon the particular ORB, the distributed programming model(s) at the client and server, and the environment in which the two are running. This includes the operating platform and the client/server roles given to WebSphere and the ORB. IBM does not warrant or make any representation as to the suitability or interoperability capabilities between WebSphere and any non-IBM, third-party ORB.

Any sample software code or other information provided by IBM that demonstrates the operation of a non-IBM ORB does not reflect endorsement of that particular ORB, or best practice use of that ORB. For further information concerning the use and operation of a non-IBM ORB, please consult the documentation provided by the vendor of such ORB.

THE INFORMATION CONTAINED IN THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION ANY SAMPLE SOFTWARE CODE PROVIDED IN CONJUNCTION WITH THIS DOCUMENT, IS PROVIDED BY IBM AND ITS AFFILIATES "AS IS." IBM AND ITS AFFILIATES MAKE NO WARRANTIES, REPRESENTATIONS OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE INFORMATION CONTAINED HEREIN, OR THE SAMPLE SOFTWARE CODE PROVIDED IN CONJUNCTION WITH THIS DOCUMENT. UNDER NO CIRCUMSTANCES SHALL IBM OR ITS AFFILIATES BE LIABLE FOR LOST SALES, LOST PROFITS, LOST SAVINGS, LOST DATA OR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND, EVEN IF INFORMED OF THE POSSIBILITY FOR SUCH DAMAGES, THAT ARE IN ANY WAY RELATED TO THE USE, OR INABILITY TO USE, THE INFORMATION CONTAINED IN THIS DOCUMENT, OR THE SAMPLE SOFTWARE CODE PROVIDED IN CONJUNCTION WITH THIS DOCUMENT.

Any party using the sample software code provided in connection with this document agrees to observe all applicable laws relating to its use, including export regulations.

Introduction

This section provides an introduction to WebSphere CORBA interoperability. The following topics are covered:

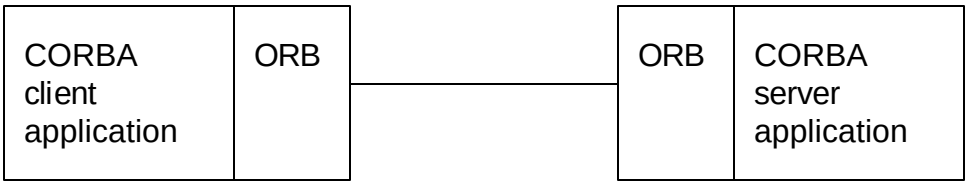
- Definition of WebSphere CORBA interoperability
- Importance of WebSphere CORBA interoperability
- Issues with WebSphere CORBA interoperability
- Examples of WebSphere CORBA interoperability

Definition of WebSphere CORBA Interoperability

In this document, the term *WebSphere CORBA interoperability* refers to the ability for a WebSphere EJB component to interact as a client or server with a CORBA application. The interaction occurs in a way that supports the CORBA programming model. Other types of WebSphere components, such as Java servlets, also can interact with a CORBA application. However, this document describes only the interoperability between a WebSphere EJB component and a CORBA application.

Definition of a CORBA Application

A *CORBA application* is a distributed client or server application that adheres to the CORBA programming model as defined by the Open Management Group (OMG) in CORBA specifications. A CORBA application can interact with other CORBA applications. The following figure illustrates a CORBA client and server application:



A CORBA application is run with a CORBA Object Request Broker (ORB). The CORBA ORB handles the protocols required for client and server interaction. A CORBA application could be run on any platform supported by the CORBA ORB. A CORBA application could be coded in any programming language that is covered in the CORBA specifications and supported by the CORBA ORB.

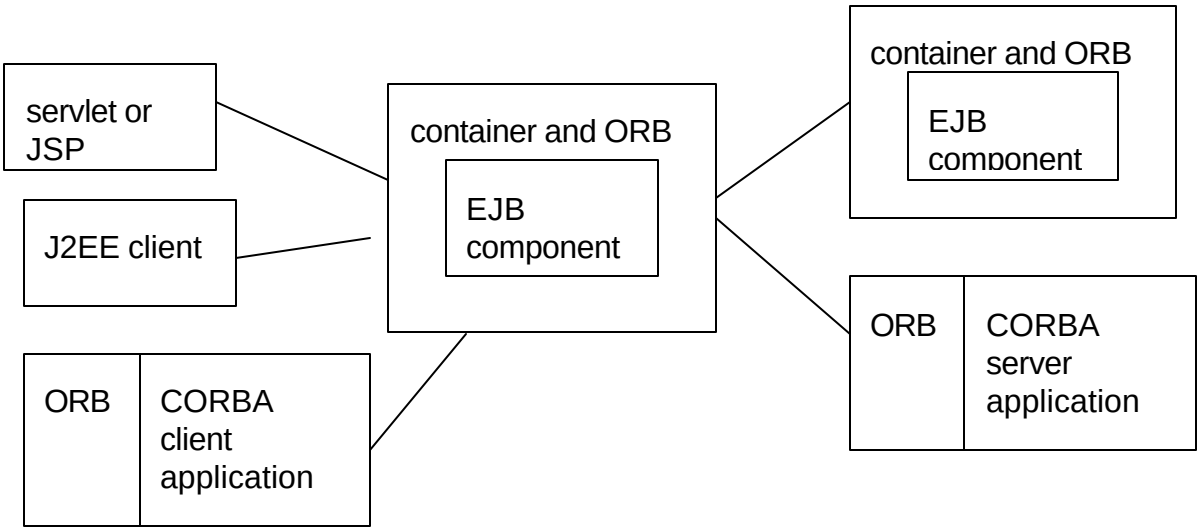
The following products provide a CORBA ORB and development environment:

- All of the version 5 IBM WebSphere Application Server products provide a CORBA ORB and a client development environment supporting the Java programming language.
- The IBM WebSphere Application Server Enterprise Version 5 product provides a CORBA ORB and a client and server development environment supporting the C++ programming language.
- Third party vendor products provide CORBA ORB and client and server development environments supporting various programming languages.

Definition of an EJB Component

An *EJB component*, as defined in this document, is a distributed Java application that adheres to the EJB programming model as specified by Sun Microsystems in Java 2 Extended Edition (J2EE) Version 1.3 or higher. A J2EE component can interact with other J2EE components including J2EE client applications, Java servlets, Java server pages (JSPs), and other EJB components. Also, an EJB component can interact with CORBA applications.

The following figure illustrates an EJB component and the J2EE components and CORBA applications with which it can interact:



An EJB component is run in a container. The container handles system functions for the EJB component and uses an underlying CORBA ORB to handle the protocols required for client and server interaction. The EJB component can directly access the CORBA ORB, if required for communicating with CORBA applications. An EJB component could be run on any platform supported by the container. An EJB component is coded in the Java programming language.

The following products provide an EJB container and development environment supporting the J2EE V1.3 specifications:

- All the version 5 IBM WebSphere Application Server products
- Third party vendor products

Definition of a WebSphere EJB Component

A *WebSphere EJB component*, as defined in this document, is an EJB component that uses the container and development environment of a version 5 WebSphere Application Server product. All the version 5 WebSphere Application Server products are fully compliant with version 1.3 of the J2EE specifications. Therefore, a WebSphere EJB component can interact with any of the following:

- Other EJB components
- Other types of J2EE components, including J2EE client applications, Java servlets, and JSPs)
- CORBA applications

Importance of WebSphere CORBA Interoperability

WebSphere CORBA interoperability provides an integration layer between all WebSphere components and existing, legacy CORBA applications. Using this integration layer, it is possible to integrate existing, legacy CORBA applications into a WebSphere environment without making any changes to the applications. The ability to do this provides the many benefits of code reuse, including code stability and savings in the cost of developing new applications.

Issues with WebSphere CORBA Interoperability

In theory, a WebSphere EJB component should be able to fully interact with any CORBA application. In practice, however, interaction with some CORBA applications is limited. This is because CORBA applications using different ORB implementations are not always able to fully interact with each other.

The following factors restrict or limit full interaction between CORBA applications using different ORB implementations:

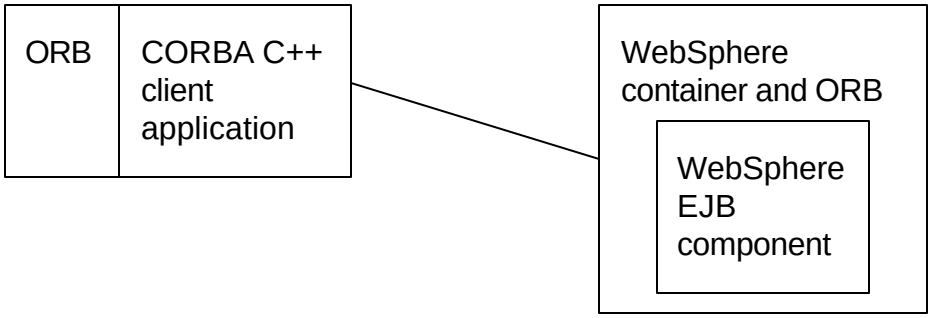
- Proprietary extensions--Some ORB implementations have added proprietary extensions to the CORBA specifications.

- Specification levels—Some ORB implementations conform to different levels of the CORBA specifications and these different levels are not always compatible with each other.
- Ambiguities--Some ORB implementations differ in the way they implement parts of the CORBA specifications because of ambiguities in the specifications.
- Bugs--Some CORBA ORBs simply have bugs.

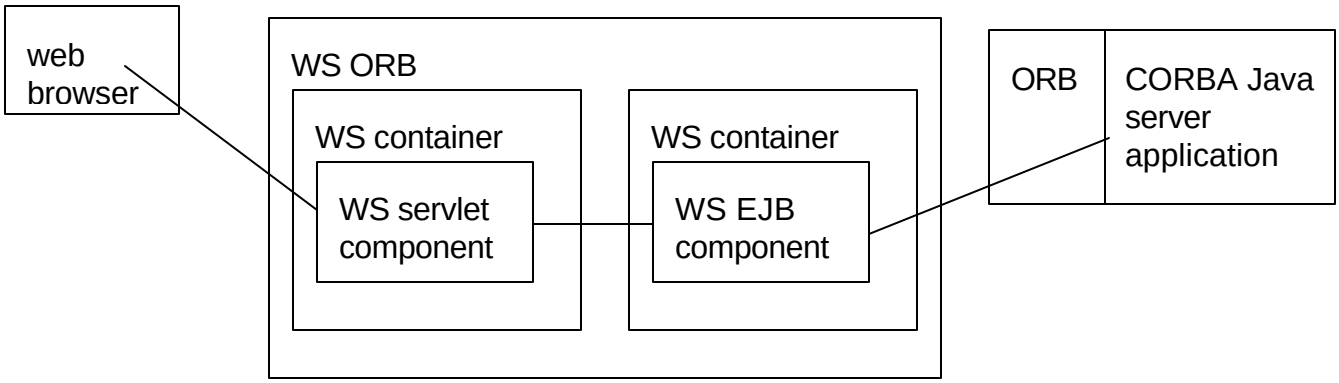
Over the long term, the CORBA specifications and ORB implementations are expected to evolve to the point where interoperability is adequate for most, if not all, scenarios. Meanwhile, it is essential to understand what interoperates as expected, what does not interoperate, and what options are available for resolving the interoperability issues.

Examples of WebSphere CORBA Interoperability

The following figure illustrates an example of WebSphere CORBA interoperability. In this example, a CORBA C++ client application invokes methods on objects residing in a WebSphere EJB component. The CORBA C++ client application is run with an ORB provided by WebSphere Application Server Enterprise Version 5. The WebSphere EJB component is run in a container provided by a version 5 WebSphere Application Server product.



The following is another example of WebSphere CORBA interoperability. In this example, a web browser uses WebSphere Java servlet and EJB components to indirectly access methods in objects residing in a CORBA Java server application. The WebSphere servlet and EJB components are run in containers provided by a version 5 WebSphere Application Server. The CORBA Java server application is run with an ORB provided by a third party vendor.



Understanding Programming Models

Before attempting to configure WebSphere CORBA interoperability, it is helpful to understand the similarities and differences between the CORBA and EJB programming models. This section provides a comparison of these programming models. The following topics are covered:

- Overview of the CORBA and EJB programming models
- CORBA programming model
- EJB programming model
- Summary of the CORBA and EJB programming models

Overview CORBA and EJB Programming Models

The CORBA and EJB programming models both make it possible for distributed clients and servers to interact with each other by agreeing on a common set of specifications. The following table lists the types of specifications that each programming model uses and the purpose of each specification type.

Specification type	**Purpose**
Object definition language	to specify a common language for defining interfaces to server objects.
Object invocation	to specify a common mechanism for invoking methods on server objects.
Communications protocol	to specify a common protocol for the client and server code to use in transmitting data over the network.
Security protocol	to specify a common protocol for securing data transmitted over the network.
Object reference syntax	to specify a common syntax for referencing server objects.
Naming service	to specify a common interface to a namespace directory.
Naming service bootstrap	to specify a common interface for bootstrapping (obtaining access to) the namespace directory.
Transaction service	to specify a common interface for requesting transactions.

CORBA Programming Model

The following describes the specifications that the CORBA programming model uses.

Object definition language

The CORBA programming model uses the CORBA Interface Definition Language (IDL) specification as a common language for defining interfaces to server objects. Typically, the developer of the server application codes the IDL definitions.

Object Invocation

The CORBA programming model uses the CORBA IDL stub and skeleton code specifications as a common mechanism for invoking methods on server objects.

Typically, the developer of the server application runs the IDL definitions through an IDL compiler to generate IDL skeleton code, in the programming language of the server application, and uses this code with the server application. Likewise, the developer of the client application runs the IDL definitions through an IDL compiler to generate IDL stub code, in the programming language of the client application, and uses this code with the client application.

The CORBA programming model also supports the specification for CORBA dynamic invocation method as an alternate mechanism for invoking methods on server objects.

Communications Protocol

The CORBA programming model uses the CORBA Interoperable Inter-ORB (IIOP) specification as a common protocol for transmitting data between IDL stub and skeleton code over a TCP/IP network. The IIOP specification is based on the CORBA General Inter-ORB (GIOP) protocol.

Security Protocol

The CORBA programming model uses the CORBA CSIv2 specification as a common protocol for transmitting data over the IIOP protocol. Typically, CSIv2 is implemented by the CORBA ORBs used by the client and server applications. The CORBA programming model also uses other security specifications, but the CSIv2 specification is the most common.

Object References

The CORBA programming model uses the CORBA Interoperable Object Reference (IOR) specification as a standard syntax for locating server objects.

Naming Service

The CORBA programming model uses the CORBA CosNaming specification as a common directory interface for storing and retrieving the locations of server objects.

The CORBA programming model uses one of two interface specifications as a common method to bootstrap the directory pointed to by the CosNaming interface. One method is for the client to configure the location of the directory as specified in the CORBA Interoperable Name Service (INS) specification and then call the CORBA resolve_initial_references API. The other method is for the client to call the string_to_object API passing in an object URL in the format specified by the CORBA INS specification.

Transaction Service

The CORBA programming model uses the CORBA OTS specification as a common interface for clients to use in requesting server transactions.

EJB Programming Model

The following describes the specifications that the EJB programming model uses.

Object Definitions

The EJB programming model uses the J2EE specifications for the EJB home and remote interface definitions, which are coded in Java, as a common language for defining interfaces to server objects. Typically, the developer of the WebSphere EJB component codes or generates these definitions.

To support CORBA interoperability, the EJB programming model also can use the specification for CORBA IDL as a common language for defining interfaces to server objects. The IDL definitions could be obtained in any of the following ways:

- The developer could code IDL definitions of the EJB home and remote interfaces
- The developer could use the WebSphere rmic compiler to generate IDL definitions from the definitions of the Java EJB home and remote interfaces
- The developer could acquire IDL definitions from the developer of a CORBA server application

Object Invocation

The EJB programming model uses the specification for RMI stub and tie code as the standard for invoking methods on server objects.

Typically, when the EJB component is deployed, the EJB home and remote definitions are run through an rmic compiler to generate RMI stub and tie code, both in the Java programming language. The RMI stub and tie code are packaged along with the EJB code in an EJB jar file. The EJB jar file is made available to the client by creating a dependency to the EJB jar file in the MANIFEST.MF file of the client JAR file.

To support CORBA interoperability, the EJB programming model also supports the CORBA specification for IDL stub code as an alternate common method for invoking methods on server objects. The developer could generate IDL stub files in Java by running IDL definitions through a Java IDL compiler (idlj).

Communications Protocol

The EJB programming model uses a combination of two specifications to transmit data between the client and server. The first specification is the CORBA IIOP version 1.2 specification, which is based on the CORBA GIOP version 1.2 specification. This specification is used to transmit data over the network. The second specification is the CORBA Java to IDL mapping specification. This specification is used to translate between Java RMI data and the data types supported by the CORBA IIOP version 1.2 specification. The second specification is not used if the client and server are using IDL code, rather than Java RMI code, as the object invocation method.

Security Service

The EJB programming model uses the CORBA specification for CSIv2 as the common protocol for securing data transmitted over the IIOP protocol.

Object References

The EJB programming model uses the CORBA specification for IOR as the common syntax for locating home references of EJB objects.

Naming Service

The EJB programming model uses the J2EE specification for the Java Naming and Directory Interface (JNDI) as the common directory interface for storing and retrieving the locations to EJB home references of EJB objects. JNDI is a front-end to the CORBA CosNaming specification. The EJB programming model uses the InitialContext interface to bootstrap the JNDI directory.

To support the CORBA programming model, the EJB programming model also supports the CosNaming directory interface and supports either resolve_initial_references with the location configured as specified by INS or string_to_object passing in an object URL in the INS format as a way to bootstrap to the CosNaming directory.

Transaction Service

The WebSphere EJB programming model uses the J2EE Java Transaction Service (JTS) specification as a standard for clients to use in requesting server transactions. The JTS specification is a Java implementation of the CORBA OTS specification, so the WebSphere EJB programming model supports the OTS interfaces.

Summary of the CORBA and EJB Programming Models

The following table summarizes the specifications that the CORBA and WebSphere EJB programming models use.

Specification Type	CORBA Programming Model	EJB Programming Model
Object definition language	IDL	Java EJB home and remote interfaces or IDL
Object invocation	IDL code	RMI code

	or dynamic method invocation	or IDL code
Communications protocol	IIOP/GIOP	IIOP/GIOP V1.2 and RMI to IDL Mapping, if required
Security protocol	CSIv2	CSIv2
Object reference	IOR	IOR
Naming service	CosNaming, using resolve_initial_references or string_to_object with an INS URL to bootstrap	JNDI, using InitialContext to bootstrap or CosNaming, using resolve_initial_references or string_to_object with an INS URL to bootstrap
Transaction service	OTS	JTS/OTS

Configuring WebSphere CORBA Interoperability

This section describes the general procedures for configuring interoperability between WebSphere EJB components and CORBA applications. The following topics are covered:

- Configuring interoperability with CORBA client applications
- Configuring interoperability with CORBA server applications

Configuring Interoperability with CORBA Client Applications

In general, you can configure interoperability between a CORBA client application and a WebSphere EJB component by generating IDL definitions and stub code from the WebSphere EJB home and remote interfaces and then calling these interfaces from the CORBA client application. The following procedures describe how to do this.

Before doing these procedures, ensure that the ORB used by the CORBA client application conforms to specifications supported by the WebSphere EJB programming model. These specifications are described in the previous section of this document. If the ORB is using interfaces or protocols that do not conform to these specifications, interoperability might not be possible.

The procedures are divided into two groups:

- Procedures to perform using WebSphere tools
- Procedures to perform using CORBA tools

Procedures to Perform Using WebSphere Tools

Perform the following procedures using the tools provided by the version 5 WebSphere Application Server product:

1. Develop and configure the WebSphere EJB component.

With one exception, do this in the normal way. The exception is that if security is required, be sure to configure an inbound security protocol that is supported by the ORB of the CORBA client application.

For example, assume the CORBA client application is using the C++ ORB provided by WebSphere Application Server Enterprise Version 5. This ORB supports the CSIv2 protocol with the restriction that only transport layer authentication is

supported. Therefore, the WebSphere EJB component needs to be configured to accept inbound requests using the CSIv2 protocol with transport layer authentication. (Transport layer authentication is SSL using public certificates.)

2. Locate the WebSphere rmic compiler.

   The rmic compiler is installed with the IBM Developer Kit in the directory <*installation_root*>/java/ibm_bin.

3. Locate the WebSphere EJB JAR file.

   The WebSphere EJB JAR file is on the system classpath (unless the developer changed this classpath). Ensure that the WebSphere EJB JAR file contains classes that can be accessed by the WebSphere rmic compiler.

4. Generate IDL definitions for WebSphere EJB interfaces.

   You need to generate IDL definitions for the WebSphere EJB home and remote interface interfaces, as well as any other interfaces that are visible to the client application. Do this by running the WebSphere rmic –idl command against the classes in the WebSphere EJB JAR file that define its home and remote interfaces, and any other necessary classes.

   For example, the following command generates IDL files named Hello.idl and HelloHome.idl:

   ```
   rmic –idl com.ibm.ejb.samples.Hello com.ibm.ejb.samples.hello.HelloHome
   ```

5. Check the IDL definitions for valuetype problems.

   When rmic generates the IDL definitions, it maps all the Java serializable data types to CORBA valuetypes. The CORBA client application must provide implementations for all these valuetypes. If this will be a problem, you might want to consider a strategy that will reduce the number of valuetype definitions in the IDL file. See the section “Valuetypes” in this document for more information.

Procedures to Perform Using CORBA Tools

Perform the following procedures using the tools provided by the CORBA ORB and development environment that the CORBA client application is using:

1. Generate IDL stub code from the IDL definitions.

   Do this by running the WebSphere IDL files against the IDL compiler supplied by the ORB that the CORBA client application is using.

2. Implement any valuetype definitions.

You need to add code to the CORBA client application that implements all valuetypes defined in the WebSphere IDL files.

The CORBA C++ development environment provided by IBM WebSphere Application Server Enterprise Version 5 provides a valuetype library containing C++ implementations of some commonly used Java classes, such as Integer, Float, Vector, and Exception. Therefore, if you are using this development environment, you can use the valuetype implementations defined in this library.

3. Implement client calls to the WebSphere EJB interfaces.

The code needs to do the following:

- Bootstrap the CosNaming service used by the WebSphere EJB component. This can be done using one of two methods:
  - Method 1: Call the CosNaming resolve_initial_references interface. This method requires the developer to configure the location of the WebSphere CosNaming service as specified by the CORBA INS specification.
  - Method 2: Call the CosNaming string_to_object interface with a parameter that uses the CORBA INS format to specify the object URL location of the WebSphere EJB CosNaming service.

  Note that the WebSphere JNDI service is a front-end to the WebSphere CosNaming service. Therefore, the location of the WebSphere CosNaming service is the same as the location of the WebSphere JNDI service.
- Get references to WebSphere EJB objects. This can be done using the CosNaming API interfaces.
- Call WebSphere EJB interfaces. This can be done using the interfaces defined in the IDL stub code.

4. Build the client application.

Do this in the normal way, making sure that you link or package the IDL stub files and the value type definitions with the client application.

5. Configure the client application.

Do this in the normal manner, noting the following:

- If the WebSphere EJB component requires security, be sure to configure the client application to use a security protocol in a manner that is consistent with the requirements of the WebSphere EJB component.

- If you used the resolve_initial_references interface to bootstrap the CosNaming service, it probably is necessary to configure the location of the CosNaming directory. The way this is done is specific to the ORB used by the CORBA client application.

Configuring Interoperability with CORBA Server Applications

In general, you can configure interoperability between a WebSphere EJB component and a CORBA server application by obtaining IDL definitions of the interfaces used by the CORBA server application, generating IDL stub code for these interfaces, and then calling the interfaces from the WebSphere EJB component. The following procedures describe how to do this.

Before doing these procedures, ensure that the ORB used by the CORBA server application conforms to CORBA specifications supported by the WebSphere EJB programming model. These specifications are described in the previous section of this document. If the ORB is using interfaces or protocols that do not conform to these specifications, interoperability might not be possible.

The following are the procedures. Perform all of these procedures using the tools and development environment provided by the version 5 WebSphere Application product.

1. Develop the server side of the WebSphere EJB component.

Do this in the normal manner. For example, assume you plan for the WebSphere EJB component to act as a server to a Java servlet. You need to code the EJB objects and interfaces that will be available for the servlet in the normal manner.

2. Obtain the IDL definitions of the CORBA server interfaces.

Normally, you obtain IDL definitions from the developer of the CORBA server application.

3. Locate an idlj compiler.

You can use any idlj compiler that conforms to the CORBA IDL to Java Mapping specifications. The idlj compiler provided by WebSphere Application Server

Version 5 and WebSphere Application Server Enterprise Version 5 is installed with the IBM Developer Kit in the directory <*installation_root*>/java/ibm_bin. The idlj compiler provided by IBM WebSphere Application Server Version 5 for z/OS is installed with the JDK SDK and, therefore, is accessible under the home directory of the JDK.

4. Generate IDL stub code from the IDL definitions.

Do this by running the IDL definitions of the CORBA application server against the idlj compiler.

5. Implement client calls to the CORBA server interfaces.

In the WebSphere EJB component, add code that implements client calls to the CORBA server interfaces. You can do this using the Java CORBA client APIs. Your implementation needs to do the following:

- Get an instance of the WebSphere ORB.

  It is strongly recommended that you use WebSphere ORB instance (the ORB instance that the WebSphere EJB container is using). This is important for two reasons. First, you will avoid the unintended inconsistencies that might occur when using different ORB instances. Second, you will ensure that the ORB instance will use the parameters that you have configured using the WebSphere administrative tool (for example, CSIv2 parameters).

  To get the WebSphere ORB instance, create a JNDI InitialContext object and look up the ORB under the name `java:comp/ORB`, as follows:

```
javax.naming.Context ctx = new javax.naming.InitialContext();
org.omg.CORBA.ORB orb =
(org.omg.CORBA.ORB)javax.rmi.PortableRemoteObject.narrow(ctx.looku
p("java:comp/ORB"),
org.omg.CORBA.ORB.class);
```

  The WebSphere ORB instance is a singleton object, shared by all J2EE components running in the same Java virtual machine process.

- Bootstrap the CosNaming service used by the CORBA application server. You can do this using the Java implementation of resolve_initial_references, which is supported by most ORBs. The following is an example:

```
...
```

```
   import org.omg.CORBA.ORB;
   import org.omg.CosNaming.NamingContext;
   import org.omg.CosNaming.NamingContextHelper;
   ...
   // Obtain ORB reference as shown in examples earlier in this
section
   ...
   org.omg.CORBA.Object obj =
_orb.resolve_initial_references("NameService");
   NamingContext initCtx = NamingContextHelper.narrow(obj);
   ...
```

Alternately, if the ORB used by the CORBA server application supports the CORBA INS specification, you can bootstrap using the Java implementation of the string_to_object interface, passing in an object URL location in the INS format, which specifies the location of the CosNaming directory. The following is an example:

```
   ...
   import org.omg.CORBA.ORB;
   import org.omg.CosNaming.NamingContext;
   import org.omg.CosNaming.NamingContextHelper;
   ...
   // Obtain ORB reference as shown in examples earlier in this
section
   ...
   org.omg.CORBA.Object obj =
orb.string_to_object("corbaloc:iiop:myhost.mycompany.com:2809/NameSer
vice");
   NamingContext initCtx = NamingContextHelper.narrow(obj);
   ...
```

- Get references to objects of the CORBA application server. You can do this using the Java implementation of the CosNaming interfaces.

- Call interfaces to objects implemented by the CORBA application server. You can do this using the interfaces defined in the IDL stub file.

6. Deploy and configure the WebSphere EJB component.

Do this in the normal manner, noting the following:

- If the CORBA server application requires security, be sure to configure the outbound CSIv2 security protocol in a manner that is consistent with the requirements of the CORBA server application.

- If you used the resolve_initial_references interface to bootstrap the CosNaming service, it is necessary to configure the location of the CosNaming directory.

Co-existence

Another possible way of configuring interoperability is to co-exist with another ORB; for example, to configure a WebSphere EJB component to co-exist with a third-party ORB. Co-existence is not supported because of the many problems that it creates.

Problems and Workarounds

This section discusses problems you might experience when attempting WebSphere CORBA interoperability and suggests workarounds for these problems. The following topics are covered:

- Unsupported data types
- Too many valuetypes
- Unsuccessful negotiation of wide character set
- Cannot bootstrap to name service

Unsupported Data Types

This problem occurs when one ORB implementation does not support a data type that another ORB implementation is using. Normally, you notice this problem during transmission. You might also notice this problem during the compilation of IDL definitions.

Explanation

This problem could occur for several reasons. The first reason is that the two ORBs are using different versions of the CORBA GIOP specification. The second reason is that one ORB is incorrectly implementing a data type defined in the GIOP specification.

The GIOP specification, on which the IIOP specification is based, defines all supported data types. However, the GIOP specification has been revised several times and each revision has added support for new data types. Most notably, version 1.2 of the GIOP specification added support for the valuetype data type, which the Java programming language requires. In addition, the there are differences between version 1.1 and version 1.2 in the wchar/wstring encoding.

Table 1 lists the data types defined in each version of GIOP. The ORBs of the version 5 WebSphere Application Server products conform to version 1.2 of the GIOP specification unless otherwise noted.

GIOP	Data Types	GIOP 1.0	GIOP 1.1	GIOP 1.2
simple data types	octet char short unsigned short long unsigned long float double boolean string	✓	✓	✓
	long long, unsigned long long	✓	✓	✓
	long double,	✓	✓	✓
	fixed	✓	✓	✓
	wchar, wstring		✓	✓
	enum	✓	✓	✓
compound data types	struct, union array, sequence	✓	✓	✓
	valuetype			✓
	CORBA::Object	✓	✓	✓
	any	✓	✓	✓
	context	✓	✓	✓
	exception	✓	✓	✓
message formats	Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError	✓	✓	✓
	Fragment		✓	✓
Bi-Directional				✓ (WAS 5.0 does not support Bi-Directional GIOP)

Table 1 – Supported Data Types

Workarounds

- The following are possible workarounds to this problem:
- Remove unsupported data types from IDL definitions
- Use a wrapper
- Use the Dynamic Invocation Interface

Remove Unsupported Data Types from IDL

In some cases, the IDL file provided by a server application might contain IDL definitions of interfaces and data that the client will never use. If any of these definitions contain data types that are not supported by the ORB of the client application, the developer of the client application can simply remove these definitions from the IDL file. After doing this, the developer needs to re-compile the IDL files to generate new IDL stub code and re-link the client application with the new IDL stub code.

Use a Wrapper

A wrapper is a thin intermediate server object that provides an alternative interface to an original server object's interface. A wrapper typically delegates its implementation to the original object. A wrapper could be a CORBA object or another Session Bean.

A wrapper can provide access to a server object, such as an EJB, whose interface contains data types not supported by the client ORB. If the issue is an ORB that does not support value types, then a wrapper might be designed and implemented as a CORBA object, using supported data types, to avoid the various extraneous valuetypes generated by Java-to-IDL compilers.

The wrapper might be the only way to get client access working for some third party ORBs.

If using a wrapper, be aware of the following:

- Management: The wrappers must be installed and managed in a CORBA server.
- Lifecycle: the container manages the target EJB's lifecycle automatically. If the wrapper is a CORBA object in the same server as the EJB, the wrapper's lifecycle must be explicitly managed in your code[1]. In this situation, it is better to put CORBA wrappers in a CORBA server and EJBs in a different server, and manage the wrapper's lifecycle at the CORBA server.
- Data type swizzling: The wrappers must convert between EJB types and IDL types, unless the client uses only primitive types. For example, EJB object references need to be converted into IDL object references. Java Serializables have to be converted into IDL equivalents as well.

[1] For Session Beans, this can be done by unexporting and destroying the wrapper when the bean is removed. Since not all beans are eventually removed, like entity beans, the unexporting and destruction of the wrapper might have to be explicitly exposed to the client's programming model.

- Hand coding: both the wrapper interface design and the wrapper object itself must be hand-coded. This means increased software design and maintenance costs.
- Security: If security is used, the security context must be manually propagated to and from the wrapper.
- Transactions: If transactions are used, the transaction must be manually propagated to and from the wrapper.

Dynamic Invocation Interface

As a last resort, the CORBA Dynamic Invocation Interface (DII) allows a client to make a call to a server without using IDL. Instead, the client makes a call by constructing the method parameters dynamically, storing them as CORBA::Any data types.

Too Many Valuetypes

This problem occurs when the IDL files generated from a WebSphere EJB component contain a high number of different valuetypes. This is a problem only from the perspective of the CORBA application developer. The developer becomes aware of this problem when viewing the number of different valuetypes in the IDL definitions and sizing the work required to implement these valuetypes.

Explanation

The WebSphere EJB interfaces are coded in the Java programming language, which supports serializable data. Serializable data is not supported by the CORBA specifications. Instead, the CORBA specifications dictate that all Java serializable data must be mapped to CORBA valuetypes, which were defined as part of the GIOP specifications. Therefore, when you use the rmic compiler to generate IDL definitions, the rmic compiler maps all the Java serializable data to valuetypes.

Therefore, the more complex or serializable data types that the EJB interfaces use, the more valuetypes are generated in the IDL definitions. This is especially problematic with entity beans, because entity beans are more likely to use complex types.

Workarounds

The workarounds for this problem is to do one of the following, as described previously in the section "Unsupported Data Types":

- Remove unused valuetypes from the IDL
- Use a wrapper

- Use dynamic invocation method

Unsuccessful negotiation of wchar codeset

This problem occurs when two ORBs are not successful in negotiating a common transmission code set for wide characters. When this problem occurs, the client or server application receives invalid wchar or wstring data or one of the following exceptions:

CODESET_INCOMPATIBLE
DATA_CONVERSION
INV_OBJREF
BAD_PARAM

Explanation

The CORBA specification imposes requirements on client and server ORBs to negotiate a common code set for transmitting wide characters (wchar and wstring). The client and server ORBs negotiate this code set by means of the following algorithm, which is documented in *The Common Object Request Broker: Architecture and Specification*, 2.3.1, *October 1999*.

```
CNCS = Client Native Code Set
CCCS = Client Conversion Code Sets
SNCS = Server Native Code Set, in IOR
SCCS = Server Conversion Code Sets, in IOR
TCS = Transmission Code Set

if (CNCS == SNCS)
     TCS = CNCS;
else
{
     if (elementOf(SNCS, CCCS))
          TCS = SNCS;     // client converts to server's
     else if (elementOf(CNCS, SCCS))
          TCS = CNCS;     // server converts to client's
     else if (intersection(CCCS, SCCS) != emptySet)
          // client/server both convert to/from TCS
          TCS = oneOf(intersection(CCCS, SCCS));
     else if (compatible(CNCS, SNCS)
          TCS = UTF-16;   // fallback code set
     else
          raise CODESET_INCOMPATIBLE exception;
}
```

If the client and server ORBs are properly adhering to this algorithm but unable to negotiate a common codeset, the two ORBs raise a CODESET_INCOMPATIBLE

or DATA_CONVERSION exception. If one of the ORBs is not properly adhering to this algorithm, the ORB that is adhering to the algorithm raises an INV_OBJREF or BAD_PARAM exception or an invalid wchar/wstring is received.

Workarounds

If the two ORBs are properly adhering to the algorithm but unable to negotiate a common codeset, there are no known workarounds. The only possible suggestion is to review the documentation for both ORBs to determine whether any special configuration could be used to help with this problem.

If the one of the ORBs is not adhering to the algorithm, one possible workaround is to set WebSphere to use a default wide character codeset. If this does not solve the problem, some other workarounds are suggested.

Setting WebSphere to Use a Default wchar transmission codeset

By default, the ORB will generate INVOBJ_REF and BAD_PARAM exceptions as directed by the CORBA specification. However, the ORB allows a default wchar transmission code set (TCS-W) to be specified. For objects providing wchar data in their interfaces the ORB will fall back on the default TCS-W value if a code set is not otherwise provided in an IOR code set component (client role) or incoming service context (server role).

The system property **`com.ibm.CORBA.ORBWCharDefault`** specifies the Java ORB's default TCS-W for the application server's JVM. By default this property is not set. It can be set to the either UCS2 or UTF16, which are the only values supported by WebSphere. Any other value will be ignored and the ORB will behave as though the property were not set.

Other Possible Workarounds

The following are some other possible workarounds. These workarounds are not CORBA-compliant and, **therefore, these workarounds are not supported**:

- Set the client and server native wide character code set to a common code set. This in itself may be sufficient to allow proper exchange of wchar information.
- Configure client ORB to accept IORs without wchar code set components. In this case the client might assume that the TCS-W is the same as it's native code set, or allow a default TCS-W to be specified.
- Configure server ORB to accept requests without wchar code set service contexts. In this case the server might assume that the TCS-W is the same as it's native code set, or allow a default TCS-W to be specified.

Problem Locating a Name Service

This problem occurs when a WebSphere EJB component cannot bootstrap the CosNaming directory of a CORBA server application. It also could occur when a CORBA client application cannot bootstrap the CosNaming directory of the WebSphere EJB component.

Explanation

WebSphere supports the CORBA INS specification as a standard way of specifying the object URL location of a CosNaming directory. When bootstrapping the CosNaming directory using resolve_initial_references, the object URL location of the directory is configured as specified in the INS specification. When bootstrapping to the CosNaming directory using string_to_object, the object URL location of the directory is specified as defined in the INS specifications.

The INS specification clearly specifies how to configure or specify the location of a naming service. However, the INS specification is new, so some ORBs do not yet support this specification.

Workarounds

The following are possible workarounds to this problem:

- Use a federated name space
- Obtain an IOR from a file
- Use coexistence (not supported)
- Bypass the name service

Use a Federated Name Space

A Federated Name Space is a group of Name Services bound together by references between Name Services (it is not necessary that every Name Service references every other Name Service, only that the paths necessary to facilitate object discovery are present, presumably according to an overall architecture and design).

A NamingContext of one Name Service would therefore contain a binding to a NamingContext of a remote Name Service. The NamingContext in the remote Name Service appears as a sub-context to the NamingContext in the first Name Service. Therefore, after bootstrapping into the first Name Service, normal CosNaming NamingContext interface methods can be used to traverse the federated name space to obtain a reference to a NamingContext from the remote Name Service. This will work provided the ORBs that support each Name Service are otherwise interoperable.

In order to initially set up a Federated Name Space, an application has to have references to NamingContexts from other Name Services. It can then bind each NameContext into the others. See the following two sections for how this might be accomplished.

Obtain IOR from a File

An IOR can be obtained by an application that already has access to a NamingContext in a remote Name Service. The application can use the **`ORB.object_to_string()`** method to generate the IOR from the NamingContext, and the IOR can then be stored in a file. The file is then made available to the client that is to access the remote Name Service. The client program can read the IOR from the file and use the **`ORB.string_to_object()`** to obtain the reference to the NamingContext. This needs to only be done once during initialization of the client, and from that point the NamingContext can be used to lookup whatever objects are needed that are bound into that Name Service.

This mechanism will work provided the ORBs for the remote Name Service and the client application are otherwise interoperable.
In general, an IOR to a NamingContext in a remote Name Service is typically fairly static. Therefore, the code that generates the file containing the IOR typically only has to be run once, thus making this a relatively simple mechanism to manage in a distributed environment. In addition, the file containing the IOR could be used to configure an ORB's initial service references. Finally, an application that initializes a Federated Name Space could use this technique to obtain the references to the NamingContexts of the Name Services it was federating.

Bypass the Naming Service

The following are some additional possible workarounds for obtaining a reference to a remote object:

- Obtain IOR to remote object and pass the server object reference directly to a client.

  It is important that an indirect IOR, as opposed to a direct IOR, is used to reference remote objects. An indirect IOR does not contain any references that are specific to the configuration of any server in the network, allowing dynamic resolution of object references to enable replica servers or network reconfiguration. Rather, an indirect IOR references an object through a Location Service Daemon (LSD), which locates the object and returns a direct IOR to the client ORB using a locate-forward message (user code is not involved, the client ORB handles the locate-forward message internally).

- Bind remote object reference into local name space: look-up an entry in the name server of a remote ORB, and rebind the reference in the name server of a local ORB. For example, one can write a utility to look-up an EJB component's home in the WebSphere name service, and bind that reference in the naming directory of a third party ORB.

Samples

This section describes the IBM CORBA Interoperability Samples. The following topics are covered:

- Description of the samples
- Design of the samples
- How to run the samples

Description of the Samples

The IBM CORBA Interoperability Samples demonstrate general principles by which WebSphere EJB components can interact with CORBA applications. They also demonstrate typical uses of the CORBA and EJB programming models, and they test various interoperability scenarios. The samples can be downloaded from the following location:

http://www.software.ibm.com/wsdd/library/samples/WASV501/corba.html

The samples include server and client code, which are run together in different scenarios. Each scenario tests the interaction between WebSphere EJB component and a CORBA application. In some scenarios, the WebSphere EJB component is the client and the CORBA application is the server. In other scenarios, the CORBA application is the client and the WebSphere EJB component is the server.

The samples test the sending and receiving of many different CORBA IDL data types over the IIOP protocol between the WebSphere EJB component and the CORBA application. The samples also test the use of a common CosNaming service. The samples do not test the use of security or transactions.

The following table lists the scenarios that the samples have tested:

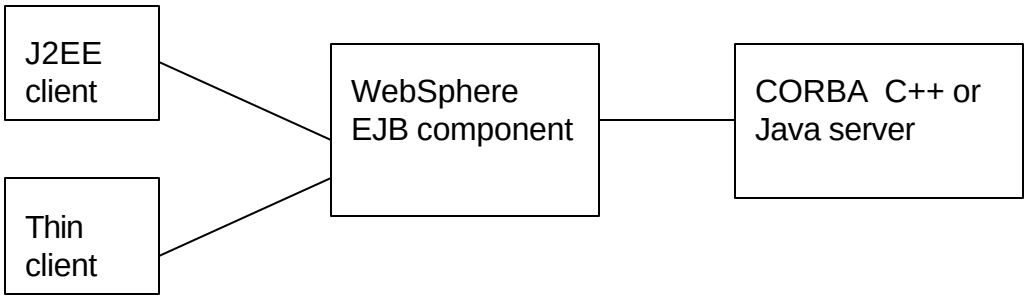
Scenario		Platforms Tested			
Client	**Server**	**Windows(R) 2000**	**AIX(R) 4.3.3**	**AIX(R) 5.1**	**Solaris(R) 2.8**
WebSphere Application Server V5 (Java)	WebSphere Corba SDK V5 (C++)	Yes	Yes	Yes	Yes
	VisiBroker 4.0 (Java)	Yes	Yes	Yes	Yes
	VisiBroker 4.1 (C++)	Yes	Yes	No	Yes
	VisiBroker 4.5.1 (Java)	Yes	Yes	Yes	Yes
	VisiBroker 4.5 (C++)	Yes	Yes	Yes	Yes
	VisiBroker 5.2.1 (Java)	Yes	Yes	Yes	Yes
	VisiBroker 5.2.1 (C++)	Yes	Yes	Yes	Yes
	OrbixE2A 5.1 (Java)	Yes	No	Yes	Yes
	OrbixE2A 5.1 (C++)	Yes	No	Yes	Yes
WebSphere Corba SDK V5 (C++)	WebSphere Application Server V5 (Java)	Yes	Yes	Yes	Yes
VisiBroker 4.1 (C++)		Yes	Yes	No	Yes
VisiBroker 4.5 (C++)		Yes	Yes	Yes	Yes
VisiBroker 5.2.1 (C++)		Yes	Yes	Yes	Yes

Design of the Samples

The samples consist of the following:

- A WebSphere EJB component, which can be used either as a client to a CORBA server application or as a server to a CORBA client application.
- A WebSphere J2EE client application. This is used only to launch the WebSphere EJB component when it is being used as a client to a CORBA application.
- A WebSphere thin client application. This is used only as an alternative way of launching the WebSphere EJB component when it is being used as a client to a CORBA application.
- CORBA C++ client applications. These are applications coded in C++ that use C++ CORBA client interfaces.
- CORBA Java client applications. These are applications coded in Java that use C++ CORBA Java interfaces.
- CORBA C++ server applications. These are applications coded in C++ that use the C++ CORBA server interfaces.
- CORBA Java server applications. These are applications coded in Java that use the CORBA Java interfaces.

The following figure illustrates the program flow when the WebSphere EJB component is the client and the CORBA application is the server. As shown in the figure, either the WebSphere J2EE client or the WebSphere thin client can be used to launch the WebSphere EJB component:



The following figure illustrates the program flow when the CORBA application is the client and the WebSphere EJB component is the server:

How to Run the Samples

To run the samples:

1. Download the samples from the URL location specified at the beginning of this section.

2. Unzip the samples.

3. Open the main web page for the samples (index.html). At the bottom of this web page, select a platform for running a scenario.

4. A second web page appears with a prompt to select the ORB for the client side of the scenario. You can select WebSphere Application Server (Java), which is the ORB of the WebSphere EJB component, or one of the CORBA applications.

5. If you select the WebSphere Application Server (Java) for the client side of the scenario, another web page appears in which you are prompted to select a CORBA ORB for the server side of the scenario. (If you select a CORBA ORB for the client side of the scenario, the server side of the scenario will be the WebSphere Application Server (Java)).

6. Another web page appears containing links to the test results for this scenario and then links for the client and server zip files.

7. Unzip the client zip file to obtain the client code and a readme file. The readme file describes how to run the client side of the scenario.

8. Unzip the server zip file to obtain the server code and a readme file. The readme file describes how to run the server side of the scenario.