BEAWebLogic
Server™

**Programming WebLogic
RMI over IIOP**

Version 8.1
Revised: December 9, 2004

Contents

About This Document

1. Overview of RMI over IIOP

2. Using RMI over IIOP Programming Models to Develop Applications

3. Configuring WebLogic Server for RMI-IIOP

A. CORBA Support for WebLogic Server 8.1

About This Document

This document explains Remote Method Invocation (RMI) over Internet Inter-ORB Protocol (IIOP) and describes how to create RMI over IIOP applications for various clients types. It describes how RMI-IIOP extends the RMI programming model by enabling Java clients to access both Java and CORBA remote objects in the BEA WebLogic Server environment.

This document is organized as follows:

- Chapter 1, "Overview of RMI over IIOP," defines RMI and RMI over IIOP, and provides general information about the WebLogic Server RMI-IIOP implementation.

- Chapter 2, "Using RMI over IIOP Programming Models to Develop Applications," describes how to develop RMI-IIOP applications using various client types.

- Chapter 3, "Configuring WebLogic Server for RMI-IIOP," describes concepts, issues, and procedures related to using WebLogic Server to support RMI-IIOP applications.

Audience

This document is written for application developers who want to enable clients to access Remote Method Invocation (RMI) remote objects using the Internet Inter-ORB Protocol (IIOP). It assumes a familiarity with the ProductName platform, CORBA, and Java programming.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server.

For more information in general about RMI over IIOP refer to the following sources.

- The OMG Web Site at http://www.omg.org/

- The Sun Microsystems, Inc. Java site at http://java.sun.com/

For more information about CORBA and distributed object computing, transaction processing, and Java, refer to the Bibliography at http://edocs.bea.com/.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number

- Your company name and company address

- Your machine type and authorization codes

- The name and version of the product you are using

- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Example*s: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

Convention	Usage
[ ]	Optional items in a syntax line. *Example*: ``` java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send] ```
|	Separates mutually exclusive choices in a syntax line. *Example*: ``` java weblogic.deploy [list|deploy|undeploy|update] password {application} {source} ```
...	Indicates one of the following in a command line: • An argument can be repeated several times in the command line. • The statement omits additional optional arguments. • You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

Overview of RMI over IIOP

The following sections provide a high-level view of RMI over IIOP:

- What Are RMI and RMI over IIOP?

- Overview of WebLogic RMI-IIOP

- Protocol Compatibility

What Are RMI and RMI over IIOP?

To understand RMI-IIOP, you should first have a working knowledge of RMI. Remote Method Invocation (RMI) is the standard for distributed object computing in Java. RMI enables an application to obtain a reference to an object that exists elsewhere in the network, and then invoke methods on that object as though it existed locally in the client's virtual machine. RMI specifies how distributed Java applications should operate over multiple Java virtual machines. RMI is written in Java and is designed exclusively for Java programs.

RMI over IIOP extends RMI to work across the IIOP protocol. This has two benefits that you can leverage. In a Java to Java paradigm this allows you to program against the standardized Internet Interop-Orb-Protocol (IIOP). If you are not working in a Java-only environment, it allows your Java programs to interact with Common Object Request Broker Architecture (CORBA) clients and execute CORBA objects. CORBA clients can be written in a variety of languages (including C++) and use the Interface-Definition-Language (IDL) to interact with a remote object.

Overview of WebLogic RMI-IIOP

RMI over IIOP is based on the RMI programming model and, to a lesser extent, the Java Naming and Directory Interface (JNDI). For detailed information on WebLogic RMI and JNDI, refer to *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs81/rmi/rmi_api.html` and *Programming with WebLogic JNDI* at `http://e-docs.bea.com/wls/docs81/jndi`. Both technologies are crucial to RMI-IIOP and it is highly recommended that you become familiar with their general concepts before starting to build an RMI-IIOP application.

The WebLogic Server 8.1 implementation of RMI-IIOP allows you to:

- Connect Java RMI clients to WebLogic Server using the standardized IIOP protocol

- Connect CORBA/IDL clients, including those written in C++, to WebLogic Server

- Interoperate between WebLogic Server and Tuxedo clients

- Connect a variety of clients to EJBs hosted on WebLogic Server

This document describes how to create applications for various clients types that use RMI and RMI-IIOP. How you develop your RMI-IIOP applications depends on what services and clients you are trying to integrate.

Figure 1-1 shows RMI Object Relationships for objects that use IIOP.

Figure 1-1   RMI Object Relationships
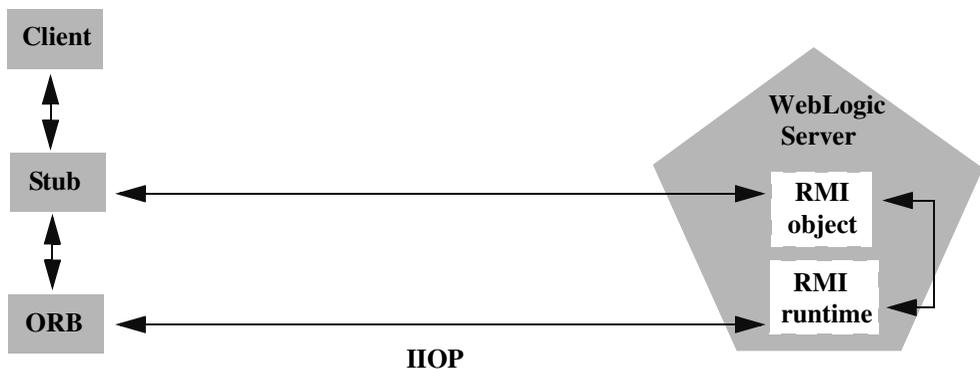
Support for RMI-IIOP with RMI (Java) Clients

You can use RMI-IIOP with Java/RMI clients, taking advantage of the standard IIOP protocol. WebLogic Server 8.1 provides multiple options for using RMI-IIOP in a Java-to-Java

environment, including the new J2EE Application Client (thin client), which is based on the new small footprint client jar. To use the new thin client, you need to have the `wlclient.jar` (located in `WL_HOME`/`server/lib`) on the client side's CLASSPATH. For more information on RMI-IIOP client options, see "Overview of RMI-IIOP Programming Models" on page 2-2.

Support for RMI-IIOP with Tuxedo Client

WebLogic Server 8.1 contains an implementation of the WebLogic Tuxedo Connector, an underlying technology that enables you to interoperate with Tuxedo servers. Using WebLogic Tuxedo Connector, you can leverage Tuxedo as an ORB, or integrate legacy Tuxedo systems with applications you have developed on WebLogic Server. For more information, see the *WebLogic Tuxedo Connector Guide* at http://e-docs.bea.com/wls/docs81/wtc.html.

Support for RMI-IIOP with CORBA/IDL Clients

The developer community requires the ability to access J2EE services from CORBA/IDL clients. However, Java and CORBA are based on very different object models. Because of this, sharing data between objects created in the two programming paradigms was, until recently, limited to Remote and CORBA primitive data types. Neither CORBA structures nor Java objects could be readily passed between disparate objects. To address this limitation, the Object Management Group (OMG) created the Objects-by-Value specification . This specification defines the enabling technology for exporting the Java object model into the CORBA/IDL programming model--allowing for the interchange of complex data types between the two models. WebLogic Server can support Objects-by-Value with any CORBA ORB that correctly implements the specification.

Protocol Compatibility

Interoperability between WebLogic Server 8.1 and WebLogic Server 6.x and 7.0 is supported in the following scenarios:

- Server-to-Server Interoperability
- Client-to-Server Interoperability

Server-to-Server Interoperability

The following table identifies supported options for achieving interoperability between two WebLogic Server instances.

Table 1-1 WebLogic Server-to-Server Interoperability

From Server	To Server → WebLogic Server 6.0	WebLogic Server 6.1 SP2 and any service pack higher than SP2	WebLogic Server 7.0	WebLogic Server 8.1
WebLogic Server 6.0	RMI/T3 HTTP	HTTP	HTTP Web Services[1]	HTTP Web Services[2]
WebLogic Server 6.1 SP2 and any service pack higher than SP2	HTTP	RMI/T3 RMI/IIOP[3] HTTP Web Services	RMI/T3 RMI/IIOP[4] HTTP Web Services	RMI/T3[5] RMI/IIOP[6] HTTP Web Services[7]
WebLogic Server 7.0	HTTP	RMI/T3 RMI/IIOP[8] HTTP	RMI/T3 RMI/IIOP[9] HTTP Web Services	RMI/T3 RMI/IIOP[10] HTTP Web Services[11]
WebLogic Server 8.1	HTTP	RMI/T3 RMI/IIOP[12] HTTP	RMI/T3 RMI/IIOP[13] HTTP Web Services[14]	RMI/T3 RMI/IIOP HTTP Web Services
Sun JDK ORB client[15]	RMI/IIOP[16]	RMI/IIOP[17]	RMI/IIOP[18]	RMI/IIOP[19]

1. Must use portable client stubs generated from the "To Server" version
2. Must use portable client stubs generated from the "To Server" version
3. No support for clustered URLs and no transaction propagation
4. No support for clustered URLs and no transaction propagation
5. Known problems with exception marshalling with releases prior to 6.1 SP4
6. No support for clustered URLs and no transaction propagation. Known problems with exception marshalling.
7. Must use portable client stubs generated from the "To Server" version
8. No support for clustered URLs and no transaction propagation
9. No support for clustered URLs
10. No support for clustered URLs

11. Must use portable client stubs generated from the "To Server" version

12.   No support for clustered URLs and no transaction propagation. Known problems with exception marshalling

13. No support for clustered URLs and no transaction propagation

14. Must use portable client stubs generated from the "To Server" version

15. This option involves calling directly into the JDK ORB from within application hosted on WebLogic Server.

16. JDK 1.3.x only. No clustering. No transaction propagation

17. JDK 1.3.x only. No clustering. No transaction propagation

18. JDK 1.3.x or 1.4.1. No clustering. No transaction propagation

19. JDK 1.3.x or 1.4.1. No clustering. No transaction propagation

Client-to-Server Interoperability

The following table identifies supported options for achieving interoperability between a stand-alone Java client application and a WebLogic Server instance.

Table 1-2 Client-to-Server Interoperability

	To Server	WebLogic Server 6.0	WebLogic Server 6.1	WebLogic Server 7.0	WebLogic Server 8.1
From Client (stand-alone)					
WebLogic Server 6.0		RMI HTTP	HTTP	HTTP Web Services[1]	HTTP Web Services[2]
WebLogic Server 6.1		HTTP	RMI/T3 HTTP Web Services	RMI/T3 HTTP Web Services[3]	RMI/T3[4] HTTP Web Services[5]
WebLogic Server 7.0		HTTP	RMI/T3 RMI/IIOP[6] HTTP	RMI/T3 RMI/IIOP[7] HTTP Web Services	RMI/T3 RMI/IIOP[8] HTTP Web Services[9]

	To Server	WebLogic Server 6.0	WebLogic Server 6.1	WebLogic Server 7.0	WebLogic Server 8.1
From Client (stand-alone)					
WebLogic Server 8.1	HTTP	RMI/T3 RMI/IIOP[10] HTTP	RMI/T3 RMI/IIOP[11] HTTP Web Services[12]	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services
Sun JDK ORB client[13]	RMI/IIOP[14]	RMI/IIOP[15]	RMI/IIOP[16]	RMI/IIOP[17]	

1. Must use portable client stubs generated from the "To Server" version
2. Must use portable client stubs generated from the "To Server" version
3. Must use portable client stubs generated from the "To Server" version
4. Known problems with exception marshalling with releases prior to 6.1 SP4
5. Must use portable client stubs generated from the "To Server" version
6. No Cluster or Failover support. No transaction propagation
7. No Cluster or Failover support
8. No Cluster or Failover support
9. Must use portable client stubs generated from the "To Server" version
10. No Cluster or Failover support and no transaction propogation. Known problems with exception marshalling
11. No Cluster or Failover support and no transaction propogation. Known problems with exception marshalling
12. Must use portable client stubs generated from the "To Server" version
13. This option involved calling directly into the JDK ORB from within a client application.
14. JDK 1.3.x only. No clustering. No transaction propagation
15. JDK 1.3.x only. No clustering. No transaction propagation
16. JDK 1.3.x or 1.4.1. No clustering. No transaction propagation
17. JDK 1.3.x or 1.4.1. No clustering. No transaction propagation

Using RMI over IIOP Programming Models to Develop Applications

The following sections describe how to use various programming models to develop RMI-IIOP applications:

- Overview of RMI-IIOP Programming Models

- ORB Implementation

- Developing a Client

- Developing a J2SE Client

- Developing a J2EE Application Client (Thin Client)

- Developing Security-Aware Clients

- Developing a WLS-IIOP Client

- Developing a CORBA/IDL Client

- Developing a WebLogic C++ Client for the Tuxedo 8.1 ORB

- RMI-IIOP Applications Using WebLogic Tuxedo Connector

- Using the CORBA API

- Using EJBs with RMI-IIOP

- Code Examples

- RMI-IIOP and the RMI Object Lifecycle

Overview of RMI-IIOP Programming Models

IIOP is a robust protocol that is supported by numerous vendors and is designed to facilitate interoperation between heterogeneous distributed systems. Two basic programming models are associated with RMI-IIOP: RMI-IIOP with RMI clients and RMI-IIOP with IDL clients. Both models share certain features and concepts, including the use of a Object Request Broker (ORB) and the Internet InterORB Protocol (IIOP). However, the two models are distinctly different approaches to creating a interoperable environment between heterogeneous systems. Simply, IIOP can be a transport protocol for distributed applications with interfaces written in either IDL or Java RMI. When you program, you must decide to use either IDL or RMI interfaces; you cannot mix them.

Several factors determine how you will create a distributed application environment. Because the different models for employing RMI-IIOP share many features and standards, it is easy to lose sight of which model you are following.

Client Types and Features

The following table lists the types of clients supported in a WebLogic Server environment, and their characteristics, features, and limitations. The table includes T3 and CORBA client options, as well as RMI-IIOP alternatives.

Table 2-1 WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements	Key Features
J2EE Application Client (thin client) (New in WLS 8.1)	RMI	Java	IIOP	WLS thin client jar JDK 1.4	Supports WLS clustering. Supports many J2EE features, including security and transactions. Supports SSL. Uses CORBA 2.4 ORB.
T3	RMI	Java	T3	full WebLogic jar	Supports WLS-Specific features. Fast, scalable. No Corba interoperability.

Client	Type	Language	Protocol	Client Class Requirements	Key Features
J2SE	RMI	Java	IIOP	no WebLogic classes	Provides connectivity to WLS environment. Does not support WLS-specific features. Does not support many J2EE features. Uses CORBA 2.3 ORB. WLInitialContextFactory is deprecated for this client in WebLogic Server 8.1. Use of com.sun.jndi.cosnaming. CNCtxFactory is required.
WLS-IIOP (Introduced in WLS 7.0)	RMI	Java	IIOP	full WebLogic jar	Supports WLS-Specific features. Supports SSL Fast, scalable. Not ORB-based.
CORBA/IDL	CORBA	Languages that OMG IDL maps to, such as C++, C, Smalltalk, COBOL	IIOP	no WebLogic classes	Uses CORBA 2.3 ORB. Does not support WLS-specific features. Does not support Java.
C++ Client	CORBA	C++	IIOP	Tuxedo libraries	Interoperability between WLS applications and Tuxedo clients/services. Supports SSL. Uses CORBA 2.3 ORB.
Tuxedo Server	CORBA or RMI	Languages that OMG IDL maps to, such as C++, C, Smalltalk, COBOL	Tuxedo-General-Inter-Orb-Protocol (TGIOP)	Tuxedo libraries	Interoperability between WLS applications and Tuxedo clients/services Uses CORBA 2.3 ORB.

ORB Implementation

WebLogic Server 8.1 provides its own ORB implementation which is instantiated by default when programs call ORB.init(), or when "java:comp/ORB" is looked up in JNDI. See "CORBA Support for WebLogic Server 8.1" on page A-1 for information how WebLogic Server complies with specifications for CORBA support in J2SE 1.4 .

Using a Foreign ORB

To use an ORB other than the default WebLogic Server implementation, set the following properties:

```
org.omg.CORBA.ORBSingletonClass=<classname>
org.omg.CORBA.ORBClass=<classname>
```

The ORBSingletonClass must be set on the server command-line. The ORBClass can be set as a property argument to ORB.init().

Using a Foreign RMI-IIOP Implementation

To use a different RMI-IIOP implementation, you must set the following two properties:

```
javax.rmi.CORBA.UtilClass=<classname>
javax.rmi.CORBA.PortableRemoteObjectClass=<classname>
```

You will get the following errors at server startup:

```
<Sep 19, 2003 9:12:03 AM CDT> <Error> <IIOP> <BEA-002015> <Using
javax.rmi.CORBA.UtilClass <classname>; The IIOP subsystem requires a
WebLogic Server-compatible UtilClass.>
```

```
<Sep 19, 2003 9:12:03 AM CDT> <Error> <IIOP> <BEA-002016> <Using
javax.rmi.CORBA.PortableRemoteObjectClass <classname>, the IIOP
subsystem requires a WebLogic Server-compatible
PortableRemoteObjectClass.>
```

indicating that the WebLogic RMI-IIOP runtime will not work.

The J2SE defaults for these properties are:

```
org.omg.CORBA.ORBSingletonClass=com.sun.corba.se.internal.corba.ORBSing
leton
org.omg.CORBA.ORBClass=com.sun.corba.se.internal.Interceptors.PIORB
javax.rmi.CORBA.UtilClass=com.sun.corba.se.internal.POA.ShutdownUtilDel
egate
javax.rmi.CORBA.PortableRemoteObjectClass=com.sun.corba.se.internal.jav
ax.rmi.PortableRemoteObject
```

Developing a Client

RMI is a Java-to-Java model of distributed computing. RMI enables an application to obtain a reference to an object that exists elsewhere in the network All RMI-IIOP models are based on RMI; however, if you follow a plain RMI model without IIOP, you cannot integrate clients written in languages other than Java. You will also be using T3, a proprietary protocol, and have WebLogic classes on your client. For information on developing RMI applications, see *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs81/rmi`.

Developing a J2SE Client

RMI over IIOP with RMI clients combines the features of RMI with the standard IIOP protocol and allows you to work completely in the Java programming language. RMI-IIOP with RMI Clients is a Java-to-Java model, where the ORB is typically a part of the JDK running on the client. Objects can be passed both by reference and by value with RMI-IIOP.

When to Use a J2SE Client

J2SE clients is oriented towards the J2EE programming model; it combines the capabilities of RMI with the IIOP protocol. If your applications are being developed in Java and you wish to leverage the benefits of IIOP, you should use the RMI-IIOP with RMI client model. Using RMI-IIOP, Java users can program with the RMI interfaces and then use IIOP as the underlying transport mechanism. The RMI client runs an RMI-IIOP-enabled ORB hosted by a J2EE or J2SE container, in most cases a 1.3 or higher JDK. Note that no WebLogic classes are required, or automatically downloaded in this scenario; this is a good way of having a minimal client distribution. You also do not have to use the proprietary t3 protocol used in normal WebLogic RMI, you use IIOP, which based on an industry, not proprietary, standard.

This client is J2SE-compliant, rather than J2EE-compliant, hence it does not support many of the features provided for enterprise-strength applications. Depending on application requirements, this client may not provide required functionality. It does not support security, transactions, or JMS.

Procedure for Developing J2SE Client

To develop an application using RMI-IIOP with an RMI client:

1. Define your remote object's public methods in an interface that extends `java.rmi.Remote`.

This remote interface may not require much code. All you need are the method signatures for methods you want to implement in remote classes.

2. Implement the interface in a class named `interfaceNameImpl` and bind it into the JNDI tree to be made available to clients.

   This class should implement the remote interface that you wrote, which means that you implement the method signatures that are contained in the interface. All the code generation that will take place is dependent on this class file. Typically, you configure your implementation class as a WebLogic startup class and include a main method that binds the object into the JNDI tree.

3. Compile the remote interface and implementation class with a java compiler. Developing these classes in a RMI-IIOP application is no different that doing so in normal RMI. For more information on developing RMI objects, see Using WebLogic RMI.

4. Run the WebLogic RMI or EJB compiler against the implementation class to generate the necessary IIOP stub. Note that it is no longer necessary to use the `-iiop` option to generate the IIOP stubs:

   ```
   $ java weblogic.rmic nameOfImplementationClass
   ```

   A stub is the client-side proxy for a remote object that forwards each WebLogic RMI call to its matching server-side skeleton, which in turn forwards the call to the actual remote object implementation. Note that the IIOP stubs created by the WebLogic RMI compiler are intended to be used with the JDK 1.3.1_01 or higher ORB. If you are using another ORB, consult the ORB vendor's documentation to determine whether these stubs are appropriate.

5. Make sure that the files you have now created -- the remote interface, the class that implements it, and the stub -- are in the CLASSPATH of the WebLogic Server.

6. Obtain an initial context.

   RMI clients access remote objects by creating an initial context and performing a lookup (see next step) on the object. The object is then cast to the appropriate type.

   In obtaining an initial context, you must use `com.sun.jndi.cosnaming.CNCtxFactory` when defining your JNDI context factory. (WLInitialContextFactory is deprecated for this client in WebLogic Server 8.1) Use `com.sun.jndi.cosnaming.CNCtxFactory` when setting the value for the "`Context.INITIAL_CONTEXT_FACTORY`" property that you supply as a parameter to new `InitialContext()`.

Note: The Sun JNDI client supports the capability to read remote object references from the namespace, but not generic Java serialized objects. This means that you can read items such as EJBHomes out of the namespace but not DataSource objects. There is also no

support for client-initiated transactions (the JTA API) in this configuration, and no support for security. In the stateless session bean RMI Client example, the client obtains an initial context as is done below:

Obtaining an InitialContext:

```
* Using a Properties object as follows will work on JDK13
* clients.

   */

  private Context getInitialContext() throws NamingException {

try {
  // Get an InitialContext
  Properties h = new Properties();
  h.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.cosnaming.CNCtxFactory");
  h.put(Context.PROVIDER_URL, url);
  return new InitialContext(h);
} catch (NamingException ne) {
  log("We were unable to get a connection to the WebLogic server at
  "+url);
  log("Please make sure that the server is running.");
  throw ne;
  }

/**

* This is another option, using the Java2 version to get an
* InitialContext.
* This version relies on the existence of a jndi.properties file in
* the application's classpath. See
* Programming WebLogic JNDI for more information

private static Context getInitialContext()
  throws NamingException

{
  return new InitialContext();
}
```

7. Modify the client code to perform the lookup in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

RMI over IIOP RMI clients differ from regular RMI clients in that IIOP is defined as the protocol when obtaining an initial context. Because of this, lookups and casts must be performed in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

You must use the `javax.rmi.PortableRemoteObject.narrow()` method in any situation where you would normally cast an object to a specific class type. A CORBA client may return an object that doesn't implement your remote interface; the narrow method is provided by your orb to convert the object so that it implements your remote interface. For example, the client code responsible for looking up the EJBean home and casting the result to the `Home` object must be modified to use the `javax.rmi.PortableRemoteObject.narrow()` as shown below:

Performing a lookup:

```
/**
 * RMI/IIOP clients should use this narrow function
 */
private Object narrow(Object ref, Class c) {
  return PortableRemoteObject.narrow(ref, c);
}

/**
 * Lookup the EJBs home in the JNDI tree
 */
private TraderHome lookupHome()
  throws NamingException
{
  // Lookup the beans home using JNDI
  Context ctx = getInitialContext();

  try {
Object home = ctx.lookup(JNDI_NAME);
return (TraderHome) narrow(home, TraderHome.class);
} catch (NamingException ne) {
log("The client was unable to lookup the EJBHome.  Please
make sure ");
log("that you have deployed the ejb with the JNDI name
"+JNDI_NAME+" on the WebLogic server at "+url);
throw ne;
  }
}

/**
 * Using a Properties object will work on JDK130
 * clients
 */
private Context getInitialContext() throws NamingException {

  try {
// Get an InitialContext
Properties h = new Properties();
h.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.cosnaming.CNCtxFactory");
h.put(Context.PROVIDER_URL, url);
```

```
return new InitialContext(h);
  } catch (NamingException ne) {
log("We were unable to get a connection to the WebLogic
server at "+url);
log("Please make sure that the server is running.");
throw ne;
  }
}
```

The `url` defines the protocol, hostname, and listen port for the WebLogic Server and is passed in as a command-line argument.

```
public static void main(String[] args) throws Exception {

  log("\nBeginning statelessSession.Client...\n");

  String url      = "iiop://localhost:7001";
```

8. Connect the client to the server over IIOP by running the client with a command like:

```
$ java -Djava.security.manager -Djava.security.policy=java.policy
  examples.iiop.ejb.stateless.rmiclient.Client iiop://localhost:7001
```

9. Set the security manager on the client:

```
java -Djava.security.manager -Djava.security.policy==java.policy
myclient
```

To narrow an RMI interface on a client the server needs to serve the appropriate stub for that interface. The loading of this class is predicated on the use of the JDK network classloader and this is **not** enabled by default. To enable it you set a security manager in the client with an appropriate java policy file. For more information on Java security, see Sun's site at http://java.sun.com/security/index.html. The following is an example of a `java.policy` file:

```
grant {

// Allow everything for now

permission java.security.AllPermission;

}
```

Developing a J2EE Application Client (Thin Client)

A J2EE application client runs on a client machine and can provide a richer user interface than can be provided by a markup language. Application clients directly access enterprise beans running in the business tier, and may, as appropriate, communicate via HTTP with servlets running in the Web tier. An application client is typically downloaded from the server, but can be installed on a client machine.

Although a J2EE application client is a Java application, it differs from a stand-alone Java application client because it is a J2EE component, hence it offers the advantages of portability to other J2EE-compliant servers, and can access J2EE services.

The WebLogic Server application client is provided as a standard client and a JMS client, packaged as two separate jar files—`wlclient.jar` and `wljmsclient.jar`— in the `WL_HOME/server/lib` subdirectory of the WebLogic Server installation directory. Each jar is about 400 KB.

The thin client is based upon the RMI-IIOP protocol stack and leverages features new to J2SE 1.4. It also requires the support of the JDK ORB. Although the thin client will work with early releases of JRE 1.4, BEA recommends using JRE 1.4.2_04 and higher. The basics of making RMI requests are handled by the JDK, enabling a significantly smaller client. Client-side development is performed using standard J2EE APIs, rather than WebLogic Server APIs.

The development process for a thin client application is the same is as for other J2EE applications. The client can leverage standard J2EE artifacts such as InitialContext, UserTransaction, and EJBs. The WebLogic Server thin client supports these values in the protocol portion of the URL—iiop, iiops, http, https, t3, and t3s—each of which can be selected by using a different URL in InitialContext. Regardless of the URL, IIOP is used. URLs with t3 or t3s use iiop and iiops respectively. Http is tunnelled iiop, https is iiop tunnelled over https.

Server-side components are deployed in the usual fashion. Client stubs can be generated at either deployment time or runtime.To generate stubs when deploying, run `appc` with the `-iiop` and `-basicClientJar` options to produce a client jar suitable for use with the thin client. Otherwise, WebLogic Server will generate stubs on demand at runtime and serve them to the client. Downloading of stubs by the client requires that a suitable security manager be installed. The thin client provides a default light-weight security manager. For rigorous security requirements, a different security manager can be installed with the command line options `-Djava.security.manager -Djava.security.policy==policyfile`. Applets use a different security manager which already allows the downloading of stubs.

The thin client jar replaces some classes in `weblogic.jar`, if both the full jar and the thin client jar are in the CLASSPATH, the thin client jar should be first in the path. Note however that `weblogic.jar` is not required to support the thin client. If desired, you can use this syntax to run with an explicit CLASSPATH:

```
java -classpath "<WL_HOME>/lib/wlclient.jar;<CLIENT_CLASSES>"
your.app.Main
```

Note: `wljmsclient.jar` has a reference to `wlclient.jar` so it is only necessary to put one or the other Jar in the CLASSPATH.

Do not put the thin-client jar in the server-side CLASSPATH.

The thin client jar contains the necessary J2EE interface classes, such as `javax.ejb`, so no other jar files are necessary on the client.

Procedure for Developing J2EE Application Client (Thin Client)

To develop a J2EE Application Client:

1. Define your remote object's public methods in an interface that extends `java.rmi.Remote`.

   This remote interface may not require much code. All you need are the method signatures for methods you want to implement in remote classes.

2. Implement the interface in a class named `interfaceNameImpl` and bind it into the JNDI tree to be made available to clients.

   This class should implement the remote interface that you wrote, which means that you implement the method signatures that are contained in the interface. All the code generation that will take place is dependent on this class file. Typically, you configure your implementation class as a WebLogic startup class and include a main method that binds the object into the JNDI tree. Here is an excerpt from the implementation class developed from the previous Ping example:

   ```
   public static void main(String args[]) throws Exception {
     if (args.length > 0)
     remoteDomain = args[0];

     Pinger obj = new PingImpl();
     Context initialNamingContext = new InitialContext();
     initialNamingContext.rebind(NAME,obj);
     System.out.println("PingImpl created and bound to "+ NAME);

   }
   ```

3. Compile the remote interface and implementation class with a java compiler. Developing these classes in a RMI-IIOP application is no different that doing so in normal RMI. For more information on developing RMI objects, see *Using WebLogic RMI*.

4. Run the WebLogic RMI or EJB compiler against the implementation class to generate the necessary IIOP stub.

Note: If you plan on downloading stubs, it is not necessary to run `rmic`.

   ```
   $ java weblogic.rmic -iiop nameOfImplementationClass
   ```

To generate stubs when deploying, run `appc` with the `-iiop` and `-clientJar` options to produce a client jar suitable for use with the thin client. Otherwise, WebLogic Server will generate stubs on demand at runtime and serve them to the client.

A stub is the client-side proxy for a remote object that forwards each WebLogic RMI call to its matching server-side skeleton, which in turn forwards the call to the actual remote object implementation.

5. Make sure that the files you have created—the remote interface, the class that implements it, and the stub—are in the CLASSPATH of the WebLogic Server.

6. Obtain an initial context.

RMI clients access remote objects by creating an initial context and performing a lookup (see next step) on the object. The object is then cast to the appropriate type.

In obtaining an initial context, you must use `weblogic.jndi.WLInitialContextFactory` when defining your JNDI context factory. Use this class when setting the value for the "`Context.INITIAL_CONTEXT_FACTORY`" property that you supply as a parameter to new `InitialContext()`.

7. Modify the client code to perform the lookup in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

RMI over IIOP RMI clients differ from regular RMI clients in that IIOP is defined as the protocol when obtaining an initial context. Because of this, lookups and casts must be performed in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

You must use the `javax.rmi.PortableRemoteObject.narrow()` method in any situation where you would normally cast an object to a specific class type. A CORBA client may return an object that doesn't implement your remote interface; the narrow method is provided by your orb to convert the object so that it implements your remote interface. For example, the client code responsible for looking up the EJBean home and casting the result to the `Home` object must be modified to use the `javax.rmi.PortableRemoteObject.narrow()` as shown below:

Performing a lookup:

```
/**
 * RMI/IIOP clients should use this narrow function
 */
private Object narrow(Object ref, Class c) {
  return PortableRemoteObject.narrow(ref, c);
}
```

```
/**
 * Lookup the EJBs home in the JNDI tree
 */
private TraderHome lookupHome()
  throws NamingException
{
  // Lookup the beans home using JNDI
  Context ctx = getInitialContext();

  try {
Object home = ctx.lookup(JNDI_NAME);
return (TraderHome) narrow(home, TraderHome.class);
} catch (NamingException ne) {
log("The client was unable to lookup the EJBHome.  Please
make sure ");
log("that you have deployed the ejb with the JNDI name
"+JNDI_NAME+" on the WebLogic server at "+url);
throw ne;
  }
}

/**
 * Using a Properties object will work on JDK130
 * clients
 */
private Context getInitialContext() throws NamingException {

  try {
// Get an InitialContext
Properties h = new Properties();
h.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
h.put(Context.PROVIDER_URL, url);
return new InitialContext(h);
  } catch (NamingException ne) {
log("We were unable to get a connection to the WebLogic
server at "+url);
log("Please make sure that the server is running.");
throw ne;
  }
}
```

The url defines the protocol, hostname, and listen port for the WebLogic Server and is passed in as a command-line argument.

```
public static void main(String[] args) throws Exception {

  log("\nBeginning statelessSession.Client...\n");

  String url = "iiop://localhost:7001";
```

8. Connect the client to the server over IIOP by running the client with a command like:

```
$ java -Djava.security.manager -Djava.security.policy=java.policy
  examples.iiop.ejb.stateless.rmiclient.Client iiop://localhost:7001
```

Developing Security-Aware Clients

You can develop Weblogic clients using the Java Authentication and Authorization Service (JAAS) and Secure Sockets Layer (SSL).

Developing Clients that Use JAAS

JAAS is the preferred method of authentication for WebLogic Server clients and provides the ability to enforce access controls based on user identity. A typical use case would be providing authentication to read or write to a file. Users requiring client certificate authentication (also referred to as two-way SSL authentication), should use JNDI authentication.You can find more information on how to implement JAAS authentication in Using JAAS Authentication in Java Clients.

Thin-client Restrictions for JAAS

WebLogic thin-client applications only supports JAAS authentication through the following classes:

- UsernamePasswordLoginModule
- Security.runAs

Developing Clients that use SSL

BEA WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between WebLogic Server clients and servers, Java clients, Web browsers, and other servers.

All SSL clients need to specify trust. Trust is a set of CA certificates that specify which trusted certificate authorities are trusted by the client. In order to establish an SSL connection, RMI clients needs to trust the certificate authorities that issued the server's digitial certificates. The location of the server's trusted CA certificate is specified when starting the RMI client.

By default, all the trusted certificate authorities available from the JDK (`...\jre\lib\security\cacerts`) are trusted by RMI clients.If the trusted CA certificate for

the server is stored in this keystore, you are all set. However, the server's trusted CA certificate can also be stored in one of the following types of trust keystores:

- Demo Trust—The trusted CA certificates in the demonstration Trust keystore (`DemoTrust.jks`) located in the `WL_HOME\server\lib directory`. In addition, the trusted CAs in the JDK cacerts keystore are trusted. To use the Demo Trust, specify the following command-line argument:

  `-Dweblogic.security.TrustKeyStore=DemoTrust`

  Optionally, use the following command-line argument to specify a password for the JDK cacerts trust keystore:

  `-Dweblogic.security.JavaStandardTrustKeystorePassPhrase=password`

  where `password` is the password for the Java Standard Trust keystore. This password is defined when the keystore is created.

- Custom Trust—A trust keystore you create. To use Custom Trust, specify the following command-line arguments:

  `weblogic.security.CustomTrustKeyStoreFileName=filename`

  This required command-line argument specifies the fully qualified path to the trust keystore

  `weblogic.security.TrustKeystoreType=CustomTrust`

  This optional command-line argument specifies the type of the keystore. Generally, this value for type is jks.

  `weblogic.security.CustomTrustKeystorePassPhrase=password`

  This optional command-line argument specifies the password defined when creating the keystore.

You can find more information on how to implement server-side SSL in Configuring RMI over IIOP with SSL.

Thin-client Restrictions for SSL

WebLogic thin-clients only supports 2-way SSL by requiring the `SSLContext` be provided by the `SECURITY_CREDENTIALS` property. For example, see the client code below:

```
.
.
.
// Get a KeyManagerFactory for KeyManagers
System.out.println("Retrieving KeyManagerFactory & initializing");
```

```
        KeyManagerFactory kmf =
        KeyManagerFactory.getInstance("SunX509","SunJSSE");
        kmf.init(ks,keyStorePassword);

    // Get and initialize an SSLContext
    System.out.println("Initializing the SSLContext");
        SSLContext sslCtx = SSLContext.getInstance("SSL");
        sslCtx.init(kmf.getKeyManagers(),null,null);

    // Pass the SSLContext to the initial context factory and get an
    // InitialContext
    System.out.println("Getting initial context");
        Hashtable props = new Hashtable();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        props.put(Context.PROVIDER_URL,
            "corbaloc:iiops:" +
             host + ":" + port +
            "/NameService");
        props.put(Context.SECURITY_PRINCIPAL,"weblogic");
        props.put(Context.SECURITY_CREDENTIALS, sslCtx);
        Context ctx = new InitialContext(props);
    .
    .
    .
```

Security Code Examples

Security samples are provided with the WebLogic Server product. The samples are located in the _SAMPLES_HOME_\server\examples\src\examples\security directory. A description of each sample and instructions on how to build, configure, and run a sample, are provided in the package-summary.html file. You can modify these code examples and reuse them.

Developing a WLS-IIOP Client

WebLogic Server supports a "fat" RMI-IIOP client referred to as the WLS-IIOP Client. The WLS-IIOP Client supports clustering.

To support WLS-IIOP clients, you must:

- have the full weblogic.jar (located in _WL_HOME_/server/lib) in the client's CLASSPATH.

- use weblogic.jndi.WLInitialContextFactory when defining your JNDI context factory. Use this class when setting the value for the

`"Context.INITIAL_CONTEXT_FACTORY"` property that you supply as a parameter to new `InitialContext()`.

Otherwise, the procedure for developing a WLS-IIOP Client is the same as the procedure described in "Developing a J2SE Client" on page 2-5.

Note: In WebLogic Server 8.1 you do not need to use the `-D weblogic.system.iiop.enableClient=true` command line option to enable client access when starting the client. By default, if you use `weblogic.jar`, `enableClient` is set to true.

Developing a CORBA/IDL Client

RMI over IIOP with CORBA/IDL clients involves an Object Request Broker (ORB) and a compiler that creates an interoperating language called IDL. C, C++, and COBOL are examples of languages that ORB's may compile into IDL. A CORBA programmer can use the interfaces of the CORBA Interface Definition Language (IDL) to enable CORBA objects to be defined, implemented, and accessed from the Java programming language.

Guidelines for Developing a CORBA/IDL Client

Using RMI-IIOP with a CORBA/IDL client enables interoperability between non-Java clients and Java objects. If you have existing CORBA applications, you should program according to the RMI-IIOP with CORBA/IDL client model. Basically, you will be generating IDL interfaces from Java. Your client code will communicate with WebLogic Server through these IDL interfaces. This is basic CORBA programming.

The following sections provide some guidelines for developing RMI-IIOP applications with CORBA/IDL clients.

For further reference see the following Object Management Group (OMG) specifications:

– Java Language Mapping to OMG IDL Specification at
   http://www.omg.org/cgi-bin/doc?formal/01-06-07

– CORBA/IIOP 2.4.2 Specification at http://www.omg.org/cgi-bin/doc?formal/01-02-33

### Working with CORBA/IDL Clients

In CORBA, interfaces to remote objects are described in a platform-neutral interface definition language (IDL). To map the IDL to a specific language, the IDL is compiled with an IDL compiler. The IDL compiler generates a number of classes such as stubs and skeletons that the client and server use to obtain references to remote objects, forward requests, and marshall

incoming calls. Even with IDL clients it is strongly recommended that you begin programming with the Java remote interface and implementation class, then generate the IDL to allow interoperability with WebLogic and CORBA clients, as illustrated in the following sections. Writing code in IDL that can be then reverse-mapped to create Java code is a difficult and bug-filled enterprise and WebLogic does not recommend doing this.

The following figure shows how IDL takes part in a RMI-IIOP model:

Figure 2-1 IDL Client (Corba object) relationships
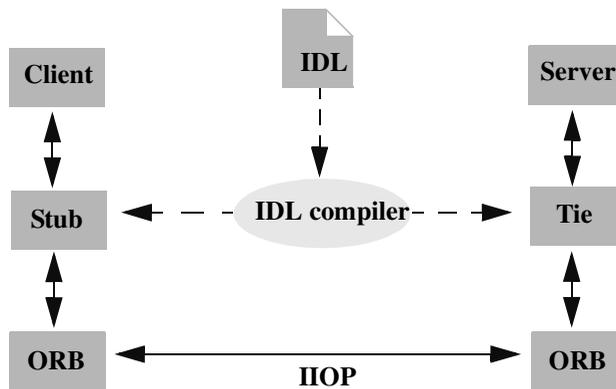
Java to IDL Mapping

In WebLogic RMI, interfaces to remote objects are described in a Java remote interface that extends `java.rmi.Remote`. The Java-to-IDL mapping specification defines how an IDL is derived from a Java remote interface. In the WebLogic RMI over IIOP implementation, you run the implementation class through the WebLogic RMI compiler or WebLogic EJB compiler with the `-idl` option. This process creates an IDL equivalent of the remote interface. You then compile the IDL with an IDL compiler to generate the classes required by the CORBA client.

The client obtains a reference to the remote object and forwards method calls through the stub. WebLogic Server implements a `CosNaming` service that parses incoming IIOP requests and dispatches them directly into the RMI runtime environment.

The following figure shows this process.

Figure 2-2 WebLogic RMI over IIOP object relationships
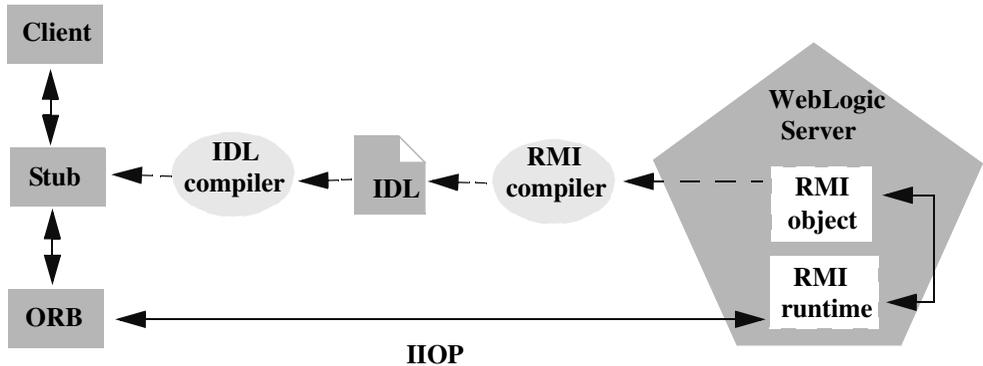
Objects-by-Value

The Objects-by-Value specification allows complex data types to be passed between the two programming languages involved. In order for an IDL client to support Objects-by-Value, you develop the client in conjunction with an Object Request Broker (ORB) that supports Objects-by-Value. To date, relatively few ORBs support Objects-by-Value correctly.

When developing an RMI over IIOP application that uses IDL, consider whether your IDL clients will support Objects-by-Value, and design your RMI interface accordingly. If your client ORB does not support Objects-by-Value, you must limit your RMI interface to pass only other interfaces or CORBA primitive data types. The following table lists ORBs that BEA Systems has tested with respect to Objects-by-Value support:

Table 2-2 ORBs Tested with Respect to Objects-by-Value Support

Vendor	Versions	Objects-by-Value
BEA	Tuxedo 8.x C++ Client ORB	supported
Borland	VisiBroker 3.3, 3.4	not supported
Borland	VisiBroker 4.x, 5.x	supported
Iona	Orbix 2000	supported (we have encountered issues with this implementation)

For more information on Objects-by-Value, see "Limitations of Passing Objects by Value" on page 3-8.

Procedure for Developing a CORBA/IDL Client

To develop an RMI over IIOP application with CORBA/IDL:

1. Follow steps 1 through 3 in "Procedure for Developing J2SE Client" on page 2-5.

2. Generate an IDL file by running the WebLogic RMI compiler for remote objects or WebLogic EJB compiler for EJBs with the `-idl` option.

   The required stub classes will be generated when you compile the IDL file. For more information on how a CORBA/IDL client can access a enterprise bean object, see "Using EJBs with RMI-IIOP" on page 2-28. Another important reference is the Java IDL specification at Java Language Mapping to OMG IDL Specification at http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm.

   The following compiler options are specific to RMI over IIOP:

Option	Function
`-idl`	Creates an IDL for the remote interface of the implementation class being compiled
`-idlDirectory`	Target directory where the IDL will be generated
`-idlFactories`	Generate factory methods for value types. This is useful if your client ORB does not support the `factory` valuetype.
`-idlNoValueTypes`	Suppresses generation of IDL for value types.
`-idlOverwrite`	Causes the compiler to overwrite an existing idl file of the same name
`-idlStrict`	Creates an IDL that adheres strictly to the Objects-By-Value specification. (not available with appc)
`-idlVerbose`	Display verbose information for IDL generation
`-idlVisibroker`	`Generate IDL somewhat compatible with Visibroker 4.1 C++`

The options are applied as shown in this example of running the RMI compiler:

> **java weblogic.rmic -idl -idlDirectory /IDL rmi_iiop.HelloImpl**

The compiler generates the IDL file within sub-directories of the `idlDirectoy` according
to the package of the implementation class. For example, the preceding command
generates a `Hello.idl` file in the `/IDL/rmi_iiop` directory. If the `idlDirectory` option
is not used, the IDL file is generated relative to the location of the generated stub and
skeleton classes.

3. Compile the IDL file to create the stub classes required by your IDL client to communicate
   with the remote class. Your ORB vendor will provide an IDL compiler.

   The IDL file generated by the WebLogic compilers contains the directives: `#include`
   `orb.idl`. This IDL file should be provided by your ORB vendor. An `orb.idl` file is
   shipped in the `/lib` directory of the WebLogic distribution. This file is only intended for
   use with the ORB included in the JDK that comes with WebLogic Server.

4. Develop the IDL client.

   IDL clients are pure CORBA clients and do not require any WebLogic classes. Depending
   on your ORB vendor, additional classes may be generated to help resolve, narrow, and
   obtain a reference to the remote class. In the following example of a client developed
   against a VisiBroker 4.1 ORB, the client initializes a naming context, obtains a reference to
   the remote object, and calls a method on the remote object.

   Code segment from C++ client of the RMI-IIOP example

```
// string to object
CORBA::Object_ptr o;

cout << "Getting name service reference" << endl;
if (argc >= 2 && strncmp (argv[1], "IOR", 3) == 0)
  o = orb->string_to_object(argv[1]);
else
  o = orb->resolve_initial_references("NameService");

// obtain a naming context
cout << "Narrowing to a naming context" << endl;
CosNaming::NamingContext_var context =
CosNaming::NamingContext::_narrow(o);
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Pinger_iiop");
name[0].kind = CORBA::string_dup("");

// resolve and narrow to RMI object
cout << "Resolving the naming context" << endl;
CORBA::Object_var object = context->resolve(name);
```

```
cout << "Narrowing to the Ping Server" << endl;
::examples::iiop::rmi::server::wls::Pinger_var ping =
  ::examples::iiop::rmi::server::wls::Pinger::_narrow(object);

// ping it
cout << "Ping (local) ..." << endl;
ping->ping();

}
```

Notice that before obtaining a naming context, initial references were resolved using the standard Object URL (CORBA/IIOP 2.4.2 Specification, section 13.6.7). Lookups are resolved on the server by a wrapper around JNDI that implements the COS Naming Service API.

The Naming Service allows Weblogic Server applications to advertise object references using logical names. The CORBA Name Service provides:

– An implementation of the Object Management Group (OMG) Interoperable Name Service (INS) specification.

– Application programming interfaces (APIs) for mapping object references into an hierarchical naming structure (JNDI in this case).

– Commands for displaying bindings and for binding and unbinding naming context objects and application objects into the namespace.

5. IDL client applications can locate an object by asking the CORBA Name Service to look up the name in the JNDI tree of WebLogic Server. In the example above, you run the client by using:

```
Client.exe -ORBInitRef
NameService=iioploc://localhost:7001/NameService.
```

Developing a WebLogic C++ Client for the Tuxedo 8.1 ORB

The WebLogic C++ client uses the Tuxedo 8.1 C++ Client ORB to generate IIOP request for EJBs running on WebLogic Server. This client supports object-by-value and the CORBA Interoperable Naming Service (INS).

When to Use a WebLogic C++ Client

You should consider using a WebLogic C++ client in the following situations:

• To simplify your development process by avoiding third-party products

• To provide a client-side solution that allows you to develop or modify existing C++ clients

Although the Tuxedo C++ Client ORB is packaged with Tuxedo 8.1 and higher, you do not need a Tuxedo license to develop WebLogic C++ clients. You can obtain a trial development copy of Tuxedo from the BEA Download Center.

How the WebLogic C++ Client works

The WebLogic C++ client using the following model to process client requests:

- The WebLogic C++ client code requests a WebLogic Server service.
    - The Tuxedo ORB generates an IIOP request.
    - The ORB object is initally instantiated and supports Object-by-Value data types.

- The Client uses the CORBA Interoperable Name Service (INS) to look up the EJB object bound to JNDI naming service. For more information on how to use the Interoperable Naming Service to get object references to initial objects such as NameService, see Interoperable Naming Service Bootstrapping Mechanism.

Figure 2-3   WebLogic C++ Client to WebLogic Server Interoperability
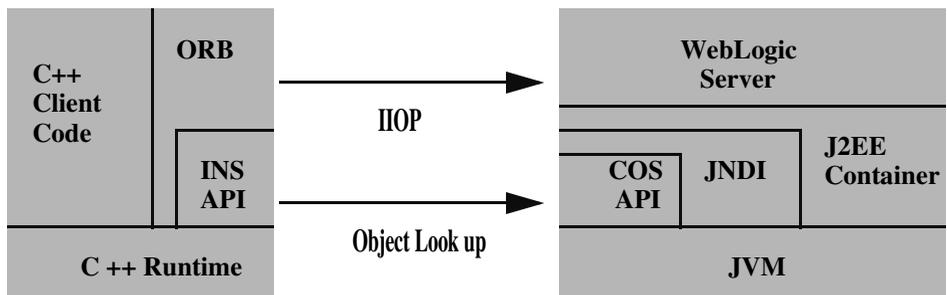
Developing WebLogic C++ Clients

Use the following steps to develop a C++ client:

1. Use the `ejbc` compiler with the `-idl` option to compile the EJB that your C++ client will interoperate with. This will generate an IDL script for the EJB.

2. Use the C++ IDL compiler to compile the IDL script and generate the CORBA client stubs, server skeletons, and header files. For information on the use of the C++ IDL Compiler, see OMG IDL Syntax and the C++ IDL Compiler.

3. Discard the server skeletons as the EJB represents the server side implementation.

4. Create a C++ client that implements an EJB as a CORBA object. For general information on how to create Corba client applications, see Creating CORBA Client Applications.

5. Use the Tuxedo `buildobjclient` command to build the client.

WebLogic C++ Client Limitations

The WebLogic C++ client has the following limitations:

- Provides security through the WebLogic Server Security service.

- Provides only server-side transaction demarcation.

WebLogic C++ Client Code Samples

WebLogic C++ client samples are provided with the WebLogic Server product. The samples are located in the `SAMPLES_HOME\server\examples\src\examples\iiop\ejb` directory. A description of each sample and instructions on how to build, configure, and run a sample, are provided in the `package-summary.html` file. You can modify these code examples and reuse them.

RMI-IIOP Applications Using WebLogic Tuxedo Connector

WebLogic Tuxedo Connector provides interoperability between WebLogic Server applications and Tuxedo services.

When to Use WebLogic Tuxedo Connector

You should consider using WebLogic Tuxedo Connector if you have developed applications on Tuxedo and are moving to WebLogic Server, or if you are seeking to integrate legacy Tuxedo systems into your newer WebLogic environment. WebLogic Tuxedo Connector allows you to leverage Tuxedo's highly scalable and reliable CORBA environment.

How the WebLogic Tuxedo Connector Works

The connector uses an XML configuration file that allows you to configure the WebLogic Server to invoke Tuxedo services. It also enables Tuxedo to invoke WebLogic Server Enterprise Java Beans (EJBs) and other applications in response to a service request.

The following documentation provides information on the Weblogic Tuxedo Connector, as well as building CORBA applications on Tuxedo:

- The WebLogic Tuxedo Connector Guide at http://e-docs.bea.com/wls/docs81/wtc.html

- For Tuxedo, CORBA topics at http://e-docs.bea.com/tuxedo/tux80/interm/corba.htm

WebLogic Tuxedo Connector Code Samples

WebLogic Tuxedo Connector IIOP samples are provided with the WebLogic Server product. The samples are located in the SAMPLES_HOME\server\examples\src\examples\iiop\ejb directory . A description of each sample and instructions on how to build, configure, and run a sample, are provided in the package-summary.html file. You can modify these code examples and reuse them.

Using the CORBA API

In WLS 8.1, the RMI-IIOP runtime has been extended to support all CORBA object types (as opposed to RMI valuetypes) and CORBA stubs. This enhancement provides the following features:

- Support of out and inout parameters

- Support for a call to a CORBA service from WebLogic Server using transactions and security.

- Support for a WebLogic ORB hosted in JNDI rather than an instance of the JDK ORB used in previous releases.

The following sections provide information on how to use the CORBA API:

- "Supporting Outbound CORBA Calls" on page 2-25

- "Using the WebLogic ORB Hosted in JNDI" on page 2-26

- "Supporting Inbound CORBA Calls" on page 2-27

- "Limitation When Using the CORBA API" on page 2-28

Supporting Outbound CORBA Calls

This section provides information on how to implement a typical development model for customers wanting to use the CORAB API for outbound calls.

1. Generate CORBA stubs from IDL using idlj, the JDKs IDL compiler.

2. Compile the stubs using javac.

3. Build EJB(s) including the generated stubs in the jar.

4. Use the WebLogic ORB hosted in JNDI to reference the external service.

Using the WebLogic ORB Hosted in JNDI

This section provides examples of several mechanisms to access the WebLogic ORB. Each of these mechanisms achieve the same effect and their constituent components can be mixed to some degree. The object returned by `narrow()` will be a CORBA stub representing the external ORB service and can be invoked on as a normal CORBA reference. Each of the following code examples assumes that the CORBA interface is call MySvc and the service is hosted at "where" in a foreign ORB's CosNaming service located at `exthost:extport`:

ORB from JNDI

```
.
.
.ORB orb = (ORB)new InitialContext().lookup("java:comp/ORB");

NamingContext nc = NamingContextHelper.narrow(orb.string_to_object("
corbaloc:iiop:exthost:extport/NameService"));

MySvc svc = MySvcHelper.narrow( nc.resolve(new NameComponent[] { new
NameComponent("where", "")}));
.
.
.
```

Direct ORB creation

```
.
.
.
ORB orb = ORB.init();

MySvc svc = MySvcHelper.narrow(orb.string_to_object("corbaname:iiop:
exthost:extport#where"));
.
.
.
```

Using JNDI

```
.
.
.
MySvc svc = MySvcHelper.narrow(new InitialContext().lookup("corbanam
e:iiop:exthost:extport#where"));
.
.
.
```

The WebLogic ORB supports most client ORB functions, including DII (Dynamic Invocation Interface). To use this support, you **must not** instantiate a foreign ORB inside the server. This will not yield any of the integration benefits of using the WebLogic ORB.

Supporting Inbound CORBA Calls

WebLogic 8.1 also provides basic support for inbound CORBA calls as an alternative to hosting an ORB inside the server. This can be achieved by using `ORB.connect()` to publish a CORBA server inside WebLogic Server. The easiest way to achieve this is to write an RMI-object which implements a CORBA interface. Given the MySVC examples above:

```
.
.
.
class MySvcImpl implements MvSvcOperations, Remote
{
        public void do_something_remote() {}


        public static main() {
                MySvc svc = new MySvcTie(this);
                InitialContext ic = new InitialContext();
                ((ORB)ic.lookup("java:comp/ORB")).connect(svc);
                ic.bind("where", svc);
        }
```

```
        }
        .
        .
        .
```

When registered as a startup class, the CORBA service will be available inside WebLogic Server's CosNaming service at the location "where".

Limitation When Using the CORBA API

CORBA Object Type support has the following limitations:

- It should not be used to make calls from one WebLogic Server instance to another WebLogic Server instance.

- It does not support clustering. If a clustered object reference is detected, WebLogic Server will use internal RMI-IIOP support to make the call. Any out or inout parameters will not be supported.

- CORBA services created by `ORB.connect()` result in a second object hosted inside the server. It is important that you use `ORB.disconnect()` to remove the object when it is no longer needed.

Using EJBs with RMI-IIOP

You can implement Enterprise JavaBeans that use RMI over IIOP to provide EJB interoperability in heterogeneous server environments:

- A Java RMI client using an ORB can access enterprise beans residing on a WebLogic Server over IIOP.

- A non-Java platform CORBA/IDL client can access any enterprise bean object on WebLogic Server.

When using CORBA/IDL clients the sources of the mapping information are the EJB classes as defined in the Java source files. WebLogic Server provides the `weblogic.appc` utility for generating required IDL files. These files represent the CORBA view into the state and behavior of the target EJB. Use the `weblogic.appc` utility to:

- Place the EJB classes, interfaces, and deployment descriptor files into a JAR file.

- Generate WebLogic Server container classes for the EJBs.

- Run each EJB container class through the RMI compiler to create stubs and skeletons.

- Generate a directory tree of CORBA IDL files describing the CORBA interface to these classes.

The `weblogic.appc` utility supports a number of command qualifiers. See "Procedure for Developing a CORBA/IDL Client" on page 2-20.

Resulting files are processed using the compiler, reading source files from the `idlSources` directory and generating CORBA C++ stub and skeleton files. These generated files are sufficient for all CORBA data types *with the exception of value types* (see "Limitations of WebLogic RMI-IIOP" on page 3-7 for more information). Generated IDL files are placed in the `idlSources` directory. The Java-to-IDL process is full of pitfalls. Refer to the Java Language Mapping to OMG IDL specification at
`http://www.omg.org/technology/documents/formal/java_language_mapping_to_om`
`g_idl.htm`. Also, Sun has an excellent guide, *Enterprise JavaBeansTM Components and CORBA Clients: A Developer Guide* at
`http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/interop.html`.

The following is an example of how to generate the IDL from a bean you have already created:

```
> java weblogic.appc -compiler javac -keepgenerated
-idl -idlDirectory idlSources
build\std_ejb_iiop.jar
%APPLICATIONS%\ejb_iiop.jar
```

After this step, compile the EJB interfaces and client application (the example here uses a CLIENT_CLASSES and APPLICATIONS target variable):

```
> javac -d %CLIENT_CLASSES% Trader.java TraderHome.java
TradeResult.java Client.java
```

Then run the IDL compiler against the IDL files built in the step where you used `weblogic.appc`, creating C++ source files:

```
>%IDL2CPP% idlSources\examples\rmi_iiop\ejb\Trader.idl
. . .
>%IDL2CPP% idlSources\javax\ejb\RemoveException.idl
```

Now you can compile your C++ client.

For an in-depth look of how EJB's can be used with RMI-IIOP see the WebLogic Server RMI-IIOP examples, located in your installation inside the
`SAMPLES_HOME`/server/examples/src/examples/iiop directory.

Code Examples

The `examples.iiop` package is included within the *WL_HOME*`/samples/examples/iiop` directory and demonstrates connectivity between numerous clients and applications. There are examples that demonstrate using EJBs with RMI-IIOP, connecting to C++ clients, and setting up interoperability with a Tuxedo Server. Refer to the example documentation for more details. For examples pertaining specifically to the Weblogic Tuxedo Connector, see the `/wlserver8.1/samples/examples/wtc` directory.

- Packaged IIOP Examples

- Additional IIOP Examples

Packaged IIOP Examples

The following table provides information on the RMI-IIOP examples provided for WebLogic Server 8.1.

Figure 2-4   WebLogic Server 8.1 IIOP Examples

Example	ORB/Protocol	Requirements
`iiop.ejb.entity.cppclient` Example provides a C++ client which calls an entity session bean in WebLogic Server.	Borland Visibroker 5.2	• Not supported for WebLogic 8.1 SP3 and higher • Specify `utf-16/iso-8859-1` as the default native codeset in the Server MBean of the `config.xml` file. • Use GIOP 1.2. Use a full corbaloc url which includes the GIOP version such as `Client -ORBInitRef NameService=corbaloc:ii op:1.2@localhost:7001/N ameService`.

Example	ORB/Protocol	Requirements
`iiop.ejb.entity.tuxclient` Example provides a Tuxedo client which uses complex valuetypes to call an entity session bean in WebLogic Server.	BEA IIOP	• Tuxedo 8.x. Does not require a Tuxedo license. • Requires custom marshalling of vector classes for Weblogic 8.1 SP3 and higher.
`iiop.ejb.entity.server.wls` Example demonstrates connectivity between a C++ client or a Tuxedo client and an entity bean.	Not Applicable	
`iiop.ejb.stateless.cppclient` Example provides a C++ CORBA client which calls a stateless session bean in WebLogic Server. The example also demonstrates how to make an outbound RMI-IIOP call to a Tuxedo server using WebLogic Tuxedo Connector.	Borland Visibroker 5.2	• Not supported for WebLogic 8.1 SP3 and higher • Specify `utf-16/iso-8859-1` as the default native codeset in the Server MBean of the `config.xml file`. • Use GIOP 1.2. Use a full corbaloc url which includes the GIOP version such as `Client -ORBInitRef NameService=corbaloc:ii op:1.2@localhost:7001/N ameService`.
`iiop.ejb.stateless.rmiclient` Example provides an RMI Java client which calls a stateless session bean in WebLogic Server. The example also demonstrates how to make an outbound RMI-IIOP call to a Tuxedo server using WebLogic Tuxedo Connector.	JDK 1.4	JDK 1.4 requires a security policy file to access server.
`iiop.ejb.stateless.sectuxcli- ent` Example illustrates a secure Tuxedo client which calls a stateless session bean from WebLogic.	BEA IIOP	Tuxedo 8.x. Does not require a Tuxedo license.

Example	ORB/Protocol	Requirements
`iiop.ejb.stateless.server.tux` Example illustrates how to call a stateless session bean from a variety of client applications through a Tuxedo Server. In conjunction with the Tuxedo Client, it also demonstrates server-to-server connectivity using WebLogic Tuxedo Connector.	Tuxedo TGIOP	• Tuxedo 8.x. • Tuxedo license Required when used with WebLogic Tuxedo Connector. • WebLogic Tuxedo Connector to provide server-to-server connectivity. See Using WebLogic Tuxedo Connector for RMI/IIOP and Corba Interoperability.
`iiop.ejb.stateless.server.wls` Example demonstrates using a variety of clients to call a stateless EJB directly in WebLogic Server or indirectly through a Tuxedo Server.	Not Applicable	
`iiop.ejb.stateless.tuxclient` Example provides a Tuxedo client which calls a stateless session bean directly in WebLogic Server or to call the same stateless session bean in WebLogic through a Tuxedo server. The example also demonstrates how to make an outbound RMI-IIOP call from a Tuxedo server to WebLogic Server using WebLogic Tuxedo Connector.	BEA IIOP	Tuxedo 8.x. Does not require a Tuxedo license.
`iiop.ejb.stateless.txtuxclient` Example provides a Tuxedo client which uses a transaction to call a stateless session bean.	BEA IIOP	Tuxedo 8.x. Does not require a Tuxedo license.

Additional IIOP Examples

The following table provides information on additional RMI-IIOP examples provided for WebLogic Server 8.1 in the dev2dev Code Library.

Figure 2-5 WebLogic Server 8.1 IIOP dev2dev Examples

Example	ORB/Protocol	Requires
`iiop.rmi.corbaclient` Example contains a CORBA client which can be used to demonstrates connectivity to a WebLogic Server.	BEA IIOP	• Tuxedo 8.0 RP56 and higher or Tuxedo 8.1. Does not require a Tuxedo license. • JDK 1.4 requires a security policy file to access server.
`iiop.rmi.cppclient` Example contains a C++ client which calls either a Tuxedo Server or a WebLogic Server. It also demonstrates server-to-server connectivity using WebLogic Tuxedo Connector.	• Borland Visibroker 5.2 • Orbix 2000	• Not supported for WebLogic 81 SP3 and higher • For Borland Visibroker 5.2: Specify `utf-16/iso-8859-1` as the default native codeset in the Server MBean of the `config.xml` file. • For Borland Visibroker 5.2: Use GIOP 1.2. Use a full corbaloc url which includes the GIOP version such as `Client -ORBInitRef NameService=corbaloc:iiop:1.2@localhost:7001/NameService`.
`iiop.rmi.rmiclient` Example provides an RMI client which demonstrates connectivity to a WebLogic Server. The example also demonstrates how to make an outbound call from WebLogic Server to a Tuxedo server using WebLogic Tuxedo Connector.	Not Applicable	Requires a security policy file to access server.
`iiop.rmi.server.tux` Example illustrates connectivity from a variety of client applications through a Tuxedo Server. In conjunction with the Tuxedo Client, it also domesticates server-to-server connectivity using WebLogic Tuxedo Connector.	Tuxedo TGIOP	• Tuxedo 8.x. • Tuxedo license Required when used with WebLogic Tuxedo Connector. • WebLogic Tuxedo Connector to provide server-to-server connectivity. See Using WebLogic Tuxedo Connector for RMI/IIOP and Corba Interoperability.

Example	ORB/Protocol	Requires
`iiop.rmi.server.wls` Example illustrates connectivity between a variety of clients, Tuxedo, and WebLogic Server using a simple Ping application.	Not Applicable	
`iiop.rmi.tuxclient` Example provides a Tuxedo client which demonstrates connectivity to a Tuxedo Server.	BEA IIOP	Tuxedo 8.x. Does not require a Tuxedo license.

RMI-IIOP and the RMI Object Lifecycle

WebLogic Server's default garbage collection causes unused and unreferenced server objects to be garbage collected. This reduces the risk running out of memory due to a large number of unused objects. This policy can lead to `NoSuchObjectException` errors in RMI-IIOP if a client holds a reference to a remote object but does not invoke on that object for a period of approximately six (6) minutes. Such exceptions should not occur with EJBs, or typically with RMI objects that are referenced by the server instance, for instance via JNDI.

The J2SE specification for RMI-IIOP calls for the use of the `exportObject()` and `unexportObject()` methods on `javax.rmi.PortableRemoteObject` to manage the lifecycle of RMI objects under RMI-IIOP, rather than Distributed Garbage Collection (DGC). Note however that `exportObject()` and `unexportObject()` have no effect with WebLogic Server's default garbage collection policy. If you wish to change the default garbage collection policy, please contact BEA technical support.

Configuring WebLogic Server for RMI-IIOP

The following sections describe concepts and procedures relating to configuring WebLogic Server for RMI-IIOP:

Set the Listening Address

To facilitate the use of IIOP, always specify a valid IP address or DNS name for the Listen Address attribute in the configuration file (`config.xml`) to listen for connections.

The Listen Address default value of `null` allows it to "listen on all configured network interfaces". However, this feature only works with the T3 protocol. If you need to configure multiple listen addresses for use with the IIOP protocol, then use the Network Channel feature, as described in "Configuring Network Resources" in *Configuring and Managing WebLogic Server*.

Setting Network Channel Addresses

The following sections provide information to consider when implementing IIOP network channel addresses for thin clients.

Considerations for Proxys and Firewalls

Many typical environments use firewalls, proxys, or other devices that hide the application server's true IP address. Because IIOP relies on a per-object addressing scheme where every object contains a host and port, anything that masks the true IP address of the server will prevent the external client from maintaining a connection. To prevent this situation, set the `PublicAddress` on the server IIOP network channel to the virtual IP that the client sees.

Considerations for Clients with Multiple Connections

IIOP clients publish addressing information that is used by the application server to establish a connection. In some situations, such as running a VPN where clients have more than one connection, the server cannot see the IP address published by the client. In this situation, you have two options:

- Use a bi-directional form of IIOP. Use the following WebLogic flag:

      -Dweblogic.corba.client.bidir=true

  In this instance, the server does not need the IP address published by the client because the server uses the inbound connection for outbound requests.

- Use the following JDK property to set the address the server uses for outbound connectons:

      -Dcom.sun.CORBA.ORBServerHost=client_*ipaddress*

  where `client_`*`ipaddress`* is an address published by the client.

Using RMI-IIOP with SSL and a Java Client

The Java clients that support SSL are the thin client and the WLS-IIOP client. To use SSL with these clients, simply specify an ssl url.

Using a IIOPS Thin Client Proxy

The IIOPs Thin Client Proxy provides a WebLogic thin client the ability to proxy outbound requests to a server. In this situation, each user routes all outbound requests through their proxy. The user's proxy then directs the request to the WebLogic Server. You should use this method when it is not practical to implement a Network Channel. To enable a proxy, set the following properties:

```
-Diiops.proxyHost=<host>

-Diiops.proxyPort=<port>
```

where:

- *hostname* is the network address of the user's proxy server.

- *port* is the port number. If not explicitly set, the value of the port number is set to 80.

- *hostname* and *port* support symbolic names, such as:

```
-Diiops.proxyHost=https.proxyHost

-Diiops.proxyPort=https.proxyPort
```

You should consider the following security implications:

- This feature does not change the behavior of WebLogic Server. However, using this feature does expose IP addresses though the client's firewall. As both ends of the connection are trusted and the linking information is encrypted, this is an acceptable security level for many environments.

- Some production environments do not allow enabling the CONNECT attribute on the proxy server. These environments should use HTTPS tunneling. For more information, see Setting Up WebLogic Server for HTTP Tunneling in *Configuring and Managing WebLogic Server.*

# Accessing WebLogic Server Objects from a CORBA Client through Delegation

WebLogic Server provides services that allow CORBA clients to access RMI remote objects. As an alternative method, you can also host a CORBA ORB (Object Request Broker) in WebLogic Server and delegate incoming and outgoing messages to allow CORBA clients to indirectly invoke any object that can be bound in the server.

Overview of Delegation

Here are the main steps to create the objects that work together to delegate CORBA calls to an object hosted by WebLogic Server.

1. Create a startup class that creates and initializes an ORB so that the ORB is co-located with the JVM that is running WebLogic Server.

2. Create an IDL (Interface Definition Language) that will create an object to accept incoming messages from the ORB.

3. Compile the IDL. This will generate a number of classes, one of which will be the Tie class. Tie classes are used on the server side to process incoming calls, and dispatch the calls to the proper implementation class. The implementation class is responsible for connecting to the server, looking up the appropriate object, and invoking methods on the object on behalf of the CORBA client.

Figure 3-1 is a diagram of a CORBA client invoking an EJB by delegating the call to an implementation class that connects to the server and operates upon the EJB. Using a similar architecture, the reverse situation will also work. You can have a startup class that brings up an ORB and obtains a reference to the CORBA implementation object of interest. This class can make itself available to other WebLogic objects throughout the JNDI tree and delegate the appropriate calls to the CORBA object.

Figure 3-1   CORBA Client Invoking an EJB with a Delegated Call

Delegate architecture
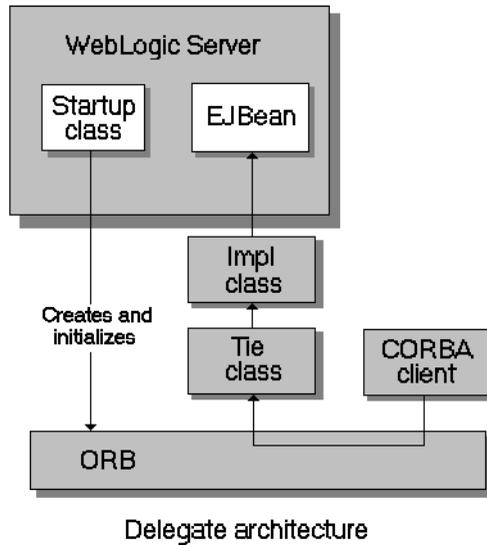
Example of Delegation

The following code example creates an implementation class that connects to the server, looks up the `Foo` object in the JNDI tree, and calls the `bar` method. This object is also a startup class that is responsible for initializing the CORBA environment by:

- Creating the ORB

- Creating the Tie object

- Associating the implementation class with the Tie object

- Registering the Tie object with the ORB

- Binding the Tie object within the ORB's naming service

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.rmi.*;
```

```
import javax.naming.*;
import weblogic.jndi.Environment;

public class FooImpl implements Foo

{
  public FooImpl() throws RemoteException {
    super();

  }

  public void bar() throws RemoteException, NamingException {
    // look up and call the instance to delegate the call to...
    weblogic.jndi.Environment env = new Environment();
    Context ctx = env.getInitialContext();
    Foo delegate = (Foo)ctx.lookup("Foo");
    delegate.bar();
    System.out.println("delegate Foo.bar called!");

  }

  public static void main(String args[]) {
    try {
      FooImpl foo = new FooImpl();

      // Create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // Create and register the tie with the ORB
      _FooImpl_Tie fooTie = new _FooImpl_Tie();
      fooTie.setTarget(foo);
      orb.connect(fooTie);

      // Get the naming context
      org.omg.CORBA.Object o = \
      orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(o);

      // Bind the object reference in naming

      NameComponent nc = new NameComponent("Foo", "");
      NameComponent path[] = {nc};
      ncRef.rebind(path, fooTie);
```

```
      System.out.println("FooImpl created and bound in the ORB
      registry.");
   }
   catch (Exception e) {
      System.out.println("FooImpl.main: an exception occurred:");
      e.printStackTrace();
   }
 }
}
```

For more information on how to implement a startup class, see *Starting and Stopping WebLogic Servers*.

Using RMI over IIOP with a Hardware LoadBalancer

Note: This feature works correctly only when the bootstrap is through a hardware load-balancer.

An optional enhancement to the WebLogic Server 8.1 BEA ORB for release service pack 3 and higher, supports hardware loadbalancing by forcing reconnection when bootstrapping. This allows hardware load-balancers to balance connection attempts

In most situations, once a connection has been established, the next NameService lookup is performed using the original connection. However, since this feature forces re-negotiation of the end point to the hardware load balancer, all in-flight requests on any existing connection are lost.

Use the `-Dweblogic.system.iiop.reconnectOnBootstrap` system property to set the connection behavior of the BEA ORB. Valid values are:

- true —Forces re-negotiation of the end point.

- false—Default value.

Environments requiring a hardware loadbalancer should set this property to true.

Limitations of WebLogic RMI-IIOP

The following sections outline various issues relating to WebLogic RMI-IIOP.

Limitations Using RMI-IIOP on the Client

Use WebLogic Server with JDK 1.3.1_01 or higher. Earlier versions are not RMI-IIOP compliant. Note the following about these earlier JDKs:

- Send GIOP 1.0 messages and GIOP 1.1 profiles in IORs.

- Do not support the necessary pieces for EJB 2.0 interoperation (GIOP 1.2, codeset negotiation, UTF-16).

- Have bugs in its treatment of mangled method names.

- Do not correctly unmarshal unchecked exceptions.

- Have subtle bugs relating to the encoding of valuetypes.

Many of these items are impossible to support both ways. Where there was a choice, WebLogic supports the spec-compliant option.

Limitations Developing Java IDL Clients

BEA Systems strongly recommends developing Java clients with the RMI client model if you are going to use RMI-IIOP. Developing a Java IDL client can cause naming conflicts and classpath problems, and you are required to keep the server-side and client-side classes separate. Because the RMI object and the IDL client have different type systems, the class that defines the interface for the server-side will be very different from the class that defines the interface on the client-side.

Limitations of Passing Objects by Value

To pass objects by value, you need to use value types (see Chapter 5 of the CORBA/IIOP 2.4.2 Specification for further information) You implement value types on each platform on which they are defined or referenced. This section describes the difficulties of passing complex value types, referencing the particular case of a C++ client accessing an Entity bean on WebLogic Server.

One problem encountered by Java programmers is the use of derived datatypes that are not usually visible. For example, when accessing an EJB finder the Java programmer will see a Collection or Enumeration, but does not pay attention to the underlying implementation because the JDK run-time will classload it over the network. However, the C++, CORBA programmer must know the type that comes across the wire so that he can register a value type factory for it and the ORB can unmarshal it.

Simply running `ejbc` on the defined EJB interfaces will **not** generate these definitions because they do not appear in the interface. For this reason `ejbc` will also accept Java classes that are not

remote interfaces--specifically for the purpose of generating IDL for these interfaces. Review the `/iiop/ejb/entity/cppclient` example to see how to register a value type factory.

Java types that are serializable but that define `writeObject()` are mapped to custom value types in IDL. You must write C++ code to unmarshal the value type manually. See *SAMPLES_HOME*`/server/src/examples/iiop/ejb/entity/tuxclient/ArrayList_i.cpp` for an example of how to do so.

Note: When using Tuxedo, you can specify the `-i` qualifier to direct the IDL compiler to create implementation files named *FileName*`_i.h` and *FileName*`_i.cpp`. For example, this syntax creates the `TradeResult_i.h` and `TradeResult_i.cpp` implementation files:

```
idl -IidlSources -i   idlSources\examples\iiop\ejb\iiop\TradeResult.idl
```

The resulting source files provide implementations for application-defined operations on a value type. Implementation files are included in a CORBA client application.

Propagating Client Identity

Until recently insufficient standards existed for propagating client identity from a CORBA client. If you have problems with client identity from foreign ORBs, you may need to implement one of the following methods:

- The identity of any client connecting over IIOP to WebLogic Server will default to `<anonymous>`. You can set the user and password in the `config.xml` file to establish a single identity for all clients connecting over IIOP to a particular instance of WebLogic Server, as shown in the example below:

```
<Server
Name="myserver"
NativeIOEnabled="true"
DefaultIIOPUser="Bob"
DefaultIIOPPassword="Gumby1234"
ListenPort="7001">
```

- You can also set the `IIOPEnabled` attribute in the `config.xml`. The default value is `"true"`; set this to `"false"` only if you want to disable IIOP support. No additional server configuration is required to use RMI over IIOP beyond ensuring that all remote objects are bound to the JNDI tree to be made available to clients. RMI objects are typically bound to the JNDI tree by a startup class. EJBean homes are bound to the JNDI tree at the time of deployment. WebLogic Server implements a `CosNaming Service` by delegating all lookup calls to the JNDI tree.

● This release supports RMI-IIOP `corbaname` and `corbaloc` JNDI references. Please refer to the CORBA/IIOP 2.4.2 Specification. One feature of these references is that you can make an EJB or other object hosted on one WebLogic Server available over IIOP to other Application Servers. So, for instance, you could add the following to your `ejb-jar.xml`:

```
<ejb-reference-description>
<ejb-ref-name>WLS</ejb-ref-name>
<jndi-name>corbaname:iiop:1.2@localhost:7001#ejb/j2ee/interop/foo</jndi
-name>
</ejb-reference-description>
```

The reference-description stanza maps a resource reference defined in `ejb-jar.xml` to the JNDI name of an actual resource available in WebLogic Server. The `ejb-ref-name` specifies a resource reference name. This is the reference that the EJB provider places within the `ejb-jar.xml` deployment file. The `jndi-name` specifies the JNDI name of an actual resource factory available in WebLogic Server.

Note: The `iiop:1.2` contained in the `<jndi-name>` section. This release contains an implementation of GIOP (General-Inter-Orb-Protocol) 1.2. The GIOP specifies formats for messages that are exchanged between inter-operating ORBs. This allows interoperability with many other ORBs and application servers. The GIOP version can be controlled by the version number in a `corbaname` or `corbaloc` reference.

These methods are not required when using `WLInitialContextFactory` in RMI clients or can be avoided by using the WebLogic C++ client as demonstrated in the `sectuxclient` example located at
SAMPLES_HOME/server/examples/src/examples/iiop/ejb/stateless/sectuxclient.

RMI-IIOP Code Examples Package

The `examples.iiop` package is in the
SAMPLES_HOME/server/examples/src/examples/iiop directory and demonstrates connectivity between numerous clients and applications. Refer to the example documentation for more details. For examples pertaining specifically to WebLogic Tuxedo Connector, see the
SAMPLES_HOME/server/examples/src/examples/wtc directory.

Additional Resources

WebLogic RMI-IIOP is intended to be a complete implementation of RMI. Please refer to the release notes for any additional considerations that might apply to your version.

● *Programming with WebLogic JNDI* at http://e-docs.bea.com/wls/docs81/jndi

● *Using WebLogic RMI* at http://e-docs.bea.com/wls/docs81/rmi

- Java Remote Method Invocation (RMI) Homepage at
  http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html

- Sun's RMI Specifications at http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html

- Sun's RMI Tutorials at

  - `http://java.sun.com/j2se/1.3/docs/guide/rmi/getstart.doc.html`

  - `http://java.sun.com/j2se/1.3/docs/guide/rmi/rmisocketfactory.doc.html`

  - `http://java.sun.com/j2se/1.3/docs/guide/rmi/activation.html`

- Sun's RMI over IIOP documentation at http://java.sun.com/products/rmi-iiop/index.html

- OMG Homepage at http://www.omg.org

- CORBA Language Mapping Specifications at
  http://www.omg.org/technology/documents/index.htm

- CORBA Technology and the Java Platform at http://java.sun.com/j2ee/corba/

- Sun's Java IDL page at http://java.sun.com/j2se/1.3/docs/guide/idl/index.html

- Objects-by-Value Specification at ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf

CORBA Support for WebLogic Server 8.1

The following sections provide the official specifications for CORBA support for this release of WebLogic Server:

- "Specification References" on page A-1

- "Supported Specification Details" on page A-2

- "Tools" on page A-2

- "Other Compatibility Information" on page A-3

Specification References

In general, this release of WebLogic Server adheres to the OMG specifications required by J2EE 1.4. For this release, the WebLogic ORB is compliant with following specification references:

- CORBA 2.3.1: formal/99-10-07 at http://www.omg.org/cgi-bin/doc?formal/99-10-07

- CORBA 2.6: formal/01-12-01 at http://www.omg.org/cgi-bin/doc?formal/01-12-01

- Revised IDL to Java language mapping: ptc/00-01-08 at http://www.omg.org/cgi-bin/doc?ptc/00-01-08

- Java to IDL language mapping: ptc/00-01-06 at http://www.omg.org/cgi-bin/doc?ptc/00-01-06

- Interoperable Naming Service: ptc/00-08-07 at http://www.omg.org/cgi-bin/doc?ptc/00-08-07

- Transaction Service 1.2.1: formal/2001-11-03 at
  http://www.omg.org/cgi-bin/doc?formal/01-11-03

Note: If the above links do not take you to the referenced specification, the OMG may have changed the URL. You can search the Object Management Group website at http://www.omg.org for the correct specification.

Supported Specification Details

Not all of the above specifications are implemented in the WebLogic ORB in this release. The following section provides a precise list of the supported specifications by chapter or section:

- CORBA 2.3.1, chapters 1-7.

- CORBA 2.3.1, sections 10.6.1 and 10.6.2 are supported for repository IDs.

- CORBA 2.3.1, section 10.7 for `TypeCode` APIs.

- CORBA 2.3.1, chapters 13 and 15 that define GIOP 1.0, 1.1, and 1.2. The WebLogic Server 8.1 ORB supports all versions of GIOP including the bi-directional GIOP feature defined in sections 15.8 and 15.9.

- The Interoperable Naming Service.

- Section 1.21.8 of the Revised IDL to Java Language Mapping Specification (ptc/00-11-03) has been changed from the version in the IDL to Java Language Mapping Specification (ptc/00-01-08).

- Transaction Service 1.2.1, as defined by the EJB 2.0 specification.

- CORBA 2.6, chapter 26, conformance level 0 plus stateful.

Tools

For this release, the WebLogic ORB is compliant with the following tools:

- The IDL to Java compiler (`idlj`) is compliant with following specification references:

  – CORBA 2.3.1, chapter 3 (IDL definition).

  – CORBA 2.3.1, chapters 5 and 6 (semantics of Value types).

  – CORBA 2.3.1, section 10.6.5 (pragmas).

  – The IDL to Java mapping specification.

– The Revised IDL to Java language mapping specification section 1.12.1 (local interfaces).

● The Java to IDL compiler (the IIOP backend for `rmic`) complies with:

– CORBA 2.3.1, chapters 5 and 6 (value types).

– The Java to IDL language mapping. Note that this implicitly references section 1.21 of the IDL to Java language mapping.

– IDL generated by the `-idl` flag complies with CORBA 2.3.1, chapter 3.

Other Compatibility Information

● The `java.util.Calendar` class is not interoperable between J2SE 1.3.x and WebLogic Server 8.1 when using RMI-IIOP. OMG issue 3151 may provide a solution for this issue but it is more likely that the solution will be implemented in a later release of the Java platform.

● The J2SE v.1.4 version of the `java.util.Calendar` class writes a `ZoneInfo` object in its `writeObject` method and calls a `readObject` to read the object. When the J2SE 1.4 `java.util.Calendar` class is serialized, it expects that if it is talking to an older J2SE version, the `ZoneInfo` object should not be present. In this situation, the stream will throw an `EOFException` exception and keep the stream position intact.