# BEA WebLogic Server™

## Programming WebLogic RMI over IIOP

Release 8.1 beta
Document Date: December 2002
Revised: December 9, 2002

## Copyright

Copyright © 2002 BEA Systems, Inc. All Rights Reserved.

## Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

## Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Programming WebLogic RMI over IIOP

| Part Number | Date | Software Version |
|---|---|---|
| N/A | December 9, 2002 | BEA WebLogic Server Version 8.1 beta |

# Contents


## About This Document

Audience ..... v
e-docs Web Site ..... vi
How to Print the Document ..... vi
Related Information ..... vi
Contact Us! ..... vii
Documentation Conventions ..... vii

## 1. Overview of RMI over IIOP

What Are RMI and RMI over IIOP? ..... 1-3
Overview of WebLogic RMI-IIOP ..... 1-4
Support for RMI-IIOP with RMI (Java) Clients ..... 1-5
Support for RMI-IIOP with CORBA/IDL Clients ..... 1-5
Support for RMI-IIOP with Tuxedo Clients ..... 1-5

## 2. Using RMI over IIOP Programming Models to Develop Applications

Overview of RMI-IIOP Programming Models ..... 2-8
Developing a T3 Client ..... 2-10
Developing a J2SE Client ..... 2-10
When to Use a J2SE Client ..... 2-10
Procedure for Developing J2SE Client ..... 2-11
Developing a J2EE Application Client (Thin Client) ..... 2-16
Procedure for Developing J2EE Application Client (Thin Client) ..... 2-17
Developing a WLS-IIOP Client ..... 2-21
Developing a CORBA/IDL Client ..... 2-22
Guidelines for Developing a CORBA/IDL Client ..... 2-22

Working with CORBA/IDL Clients ..... 2-22
Java to IDL Mapping ..... 2-23
Objects-by-Value ..... 2-24
Procedure for Developing a CORBA/IDL Client ..... 2-25
Developing Tuxedo Clients ..... 2-28
WebLogic Tuxedo Connector ..... 2-28
BEA WebLogic C++ Client ..... 2-29
Using EJBs with RMI-IIOP ..... 2-29

## 3. Configuring WebLogic Server for RMI-IIOP

Configuration Overview ..... 3-25
Using RMI-IIOP with SSL and a Java Client ..... 3-26
Accessing WebLogic Server Objects from a CORBA Client through Delegation 3-27
Overview of Delegation ..... 3-27
Example of Delegation ..... 3-28
Limitations of WebLogic RMI-IIOP ..... 3-30
Limitations Using RMI-IIOP on the Client ..... 3-30
Limitations Developing Java IDL Clients ..... 3-31
Limitations of Passing Objects by Value ..... 3-31
RMI-IIOP Code Examples Package ..... 3-32
Additional Resources ..... 3-32

# 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. |

| Convention | Usage |
|---|---|
| `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.<br>*Examples*:<br>`import java.util.Enumeration;`<br>`chmod u+w *`<br>`config/examples/applications`<br>`.java`<br>`config.xml`<br>`float` |
| *`monospace italic text`* | Variables in code.<br>*Example*:<br>`String CustomerName;` |
| UPPERCASE TEXT | Device names, environment variables, and logical operators.<br>*Examples*:<br>LPT1<br>BEA_HOME<br>OR |
| `{ }` | A set of choices in a syntax line. |
| `[ ]` | Optional items in a syntax line. *Example*:<br>`java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]` |
| `\|` | Separates mutually exclusive choices in a syntax line. *Example*:<br>`java weblogic.deploy [list\|deploy\|undeploy\|update] password {application} {source}` |
| `...` | Indicates one of the following in a command line:<br>■ An argument can be repeated several times in the command line.<br>■ The statement omits additional optional arguments.<br>■ You can enter additional parameters, values, or other information |

| Convention | Usage |
| --- | --- |
| .<br>.<br>. | Indicates the omission of items from a code example or from a syntax line. |

CHAPTER

# 1 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

## 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

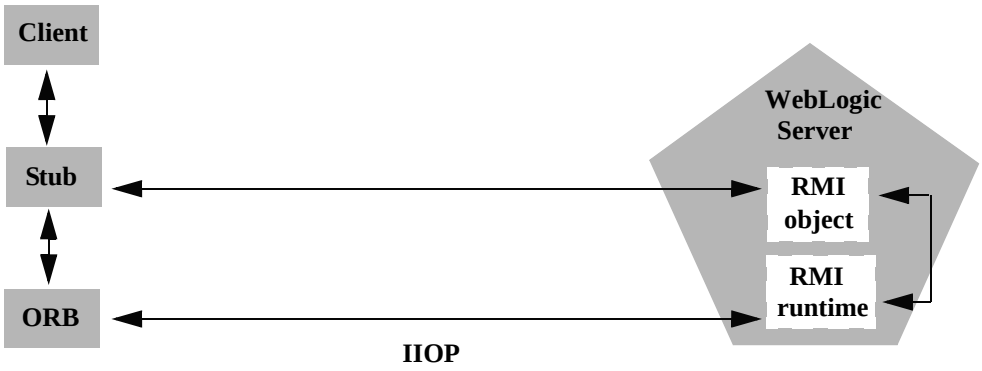
RMI over IIOP is based on the RMI programming model and, to a lesser extent, the Java Naming and Directory Interface (JNDI). For detailed information on WebLogic RMI and JNDI, refer to *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs81b/rmi/rmi_api.html` and *Programming with WebLogic JNDI* at `http://e-docs.bea.com/wls/docs81b/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.

The following figure shows an RMI Object Relationships that uses IIOP



# 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 2-8.

# 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.

# Support for RMI-IIOP with Tuxedo Clients

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/docs70/wtc.html`

CHAPTER

# 2 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
- Developing a T3 Client
- Developing a J2SE Client
- Developing a J2EE Application Client (Thin Client)
- Developing a WLS-IIOP Client
- Developing a CORBA/IDL Client
- Developing Tuxedo Clients
- Using EJBs with RMI-IIOP

# 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.

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<br>(thin client)<br>(New in WLS 8.1) | RMI | Java | IIOP | WLS thin client jar<br>JDK 1.4 | Supports WLS clustering.<br>Supports many J2EE features, including security and transactions.<br>Supports SSL.<br>Uses CORBA 2.4 ORB. |
| T3 | RMI | Java | T3 | full WebLogic jar | Supports WLS-Specific features.<br>Fast, scalable.<br>No Corba interoperability. |

| Client | Type | Language | Protocol | Client Class Requirements | Key Features |
|---|---|---|---|---|---|
| J2SE | RMI | Java | IIOP | no WebLogic classes | Provides connectivity to WLS environment.<br>Does not support WLS-specific features. Does not support many J2EE features.<br>Uses CORBA 2.3 ORB.<br>WLInitialContextFactory is deprecated for this client in WebLogic Server 8.1. Use of com.sun.jndi.cosnaming. CNCtxFactory is required. |
| WLS-IIOP<br>(Introduced in WLS 7.0) | RMI | Java | IIOP | full WebLogic jar | Supports WLS-Specific features.<br>Supports SSL<br>Fast, scalable.<br>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.<br>Does not support WLS-specific features.<br>Does not support Java. |
| C++ Client | CORBA | C++ | IIOP | Tuxedo libraries | Interoperability between WLS applications and Tuxedo clients/services.<br>Supports SSL.<br>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<br>(TGIOP) | Tuxedo libraries | Interoperability between WLS applications and Tuxedo clients/services<br>Uses CORBA 2.3 ORB. |

# Developing a T3 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/docs81b/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. For example, with the Ping example included in your Weblogic installation *SAMPLES_HOME*`/server/src/examples/iiop/rmi/server/wls`:

   ```
   public interface Pinger extends java.rmi.Remote {
   public void ping() throws java.rmi.RemoteException;
   public void pingRemote() throws java.rmi.RemoteException;
   public void pingCallback(Pinger toPing) throws
   java.rmi.RemoteException;
   }
   ```

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 that it is no longer necessary to use the `-iiop` option to generate the IIOP stubs:

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

   In the case of the Pinger example, the *`nameOfImplementationClass`* is `examples.iiop.rmi.server.wls.PingerImpl`.

   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.

For example, in the RMI client stateless session bean example (the `examples.iiop.ejb.stateless.rmiclient` package included in your distribution), an RMI client creates an initial context, performs a lookup on the EJBean home, obtains a reference to an EJBean, and calls methods on the EJBean.

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 `/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 available in JDK 1.4.n. 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 standared 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 `-clientJar` options to produce a client jar suitable for use

with the thin client. Otherwsie, WebLogic Server will generate stubs on demand at runtimeand 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 lightp-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 WebLogic thin client jar leverages features new to J2SE 1.4, so the JRE 1.4 is required. Although the thin-client will work with JRE 1.4.0, use of JRE 1.4.1_02 is recommended, due to bug fixes that affect the thin client.

**Note:** Long running clients should use JRE 1.4.1_03 when it is released.

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 neccessary J2EE interface classes, such as `javax.ejb`, so no other jar files are neccessary 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. For example, with the Ping example included in your Weblogic installation:

*SAMPLES_HOME*/server/src/examples/iiop/rmi/server/wls:

```
public interface Pinger extends java.rmi.Remote {
public void ping() throws java.rmi.RemoteException;
public void pingRemote() throws java.rmi.RemoteException;
public void pingCallback(Pinger toPing) throws
java.rmi.RemoteException;
}
```

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 donloading stubs, it is not necessary to run `rmic`.

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

In the case of the Pinger example, the *`nameOfImplementationClass`* is `examples.iiop.rmi.server.wls.PingerImpl`.

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.

   For example, in the RMI client stateless session bean example (the `examples.iiop.ejb.stateless.rmiclient` package included in your distribution), an RMI client creates an initial context, performs a lookup on the EJBean home, obtains a reference to an EJBean, and calls methods on the EJBean.

   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 a WLS-IIOP Client

In WebLogic Server 7.0, support for a "fat" RMI-IIOP client—referred to as the WLS-IIOP Client—was introduced. 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 2-10.

**Note:** In WebLogic Server 8.1 you do not need to use the `-D weblogic.system.iiop.enableClient=true` command line option to 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://cgi.omg.org/cgi-bin/doc?ptc/00-01-06`
- CORBA/IIOP 2.4.2 Specification at `http://cgi.omg.org/cgi-bin/doc?formal/99-10-07`

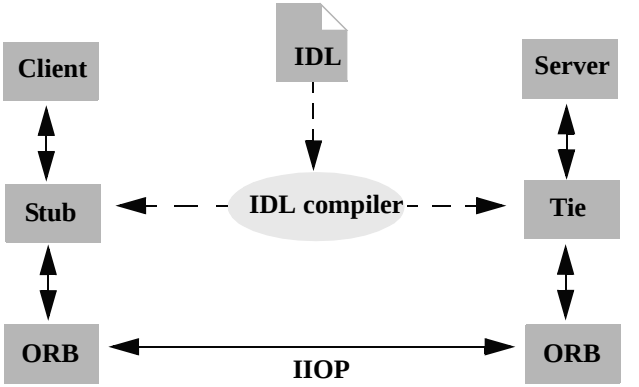
### Working with CORBA/IDL Clients

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**



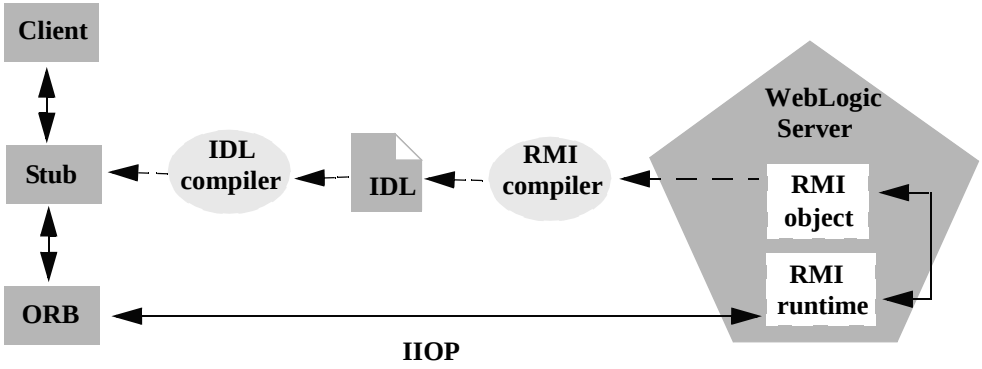

## Java to IDL Mapping

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 |
|---|---|---|
| Inprise | VisiBroker 3.3, 3.4 | not supported |
| Inprise | VisiBroker 4.x, 5.0 | supported |
| JavaSoft | JDK 1.2 | not supported |
| JavaSoft | JDK 1.3.1_01 and higher | supported |

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

| Vendor | Versions | Objects-by-Value |
|---|---|---|
| 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 3-31.

# 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 2-11.
2. Generate an IDL file by running the WebLogic RMI compiler or WebLogic EJB compiler with the `-idl` option.

   The required stub classes will be generated when you compile the IDL file. For general information on the these compilers, refer to Using WebLogic RMI and BEA WebLogic Server Enterprise JavaBeans. Also reference the Java IDL specification at Java Language Mapping to OMG IDL Specification `at http://www.omg.org/cgi-bin/doc?formal/01-06-07.`

   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. |

| Option | Function |
|---|---|
| `-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 Tuxedo Clients

WebLogic Server provides the ability to interoperate between WebLogic Server applications and Tuxedo services using RMI-IIOP. This includes calling EJBs and other applications on WebLogic from Tuxedo clients as well as other features.

The RMI-IIOP examples included in the `samples/examples/iiop` directory of your installation contain some samples of how to configure and set up your WebLogic Server to work with Tuxedo Servers and Tuxedo Clients.

## WebLogic Tuxedo Connector

WebLogic Tuxedo Connector provides interoperability between WebLogic Server applications and Tuxedo services. 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. 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, the WebLogic Tuxedo Connector allows you to leverage Tuxedo's highly scalable and reliable CORBA environment.

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

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

## BEA WebLogic C++ Client

WebLogic Server 8.1 interoperates with the Tuxedo 8.0 C++ Client ORB. This client supports object by value and the CORBA Interoperable Naming Service (INS). Tuxedo release 8.0 RP 56 and above is required. WebLogic Server users should contact their BEA Service Representative for information on how to obtain the Tuxedo C++ Client ORB.

The following documentation provides information on how to use the WebLogic C++ Client with the Tuxedo C++ Client ORB:

- For general information on how to create Tuxedo Corba client applications, see Creating CORBA Client Applications.
- For information on the use of the C++ IDL Compiler, see OMG IDL Syntax and the C++ IDL Compiler.
- For 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.

# 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 2-25.

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 3-30 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_omg_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/src/examples/iiop directory.

# 3 Configuring WebLogic Server for RMI-IIOP

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


- Configuration Overview
- Using RMI-IIOP with SSL and a Java Client
- Accessing WebLogic Server Objects from a CORBA Client through Delegation
- Limitations of WebLogic RMI-IIOP
- RMI-IIOP Code Examples Package
- Additional Resources


## Configuration Overview

Because insufficient standards exist for propagating client identity from a CORBA client, the identity of any client connecting over IIOP to WebLogic Server will default to "guest." 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.

WebLogic Server 7.0 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. WebLogic Server 7.0 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 interoperatability with many other ORBs and application servers. The GIOP version can be controlled by the version number in a `corbaname` or `corbaloc` reference.

# Using RMI-IIOP with SSL and a Java Client

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

# Accessing WebLogic Server Objects from a CORBA Client through Delegation

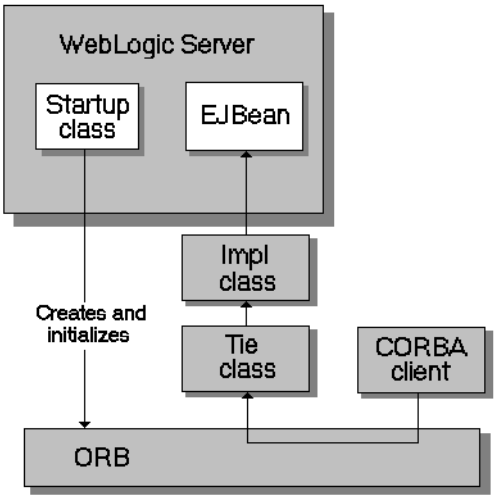
WebLogic Server provides services that allow CORBA clients to access RMI remote objects. As an alternative method, you can also host a CORBA ORB (Object Request Broker) in WebLogic Server and delegate incoming and outgoing messages to allow CORBA clients to indirectly invoke any object that can be bound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

The following is a diagram of a CORBA client invoking an EJBean by delegating the call to an implementation class that connects to the server and operates upon the EJBean. 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 througout the JNDI tree and delegate the appropriate calls to the CORBA object.




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*.

# 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 seaparate. 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 specification at `http://cgi.omg.org/cgi-bin/doc?formal/99-10-07` 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 (see the *SAMPLES_HOME*`/server/src/examples/iiop/ejb/entity/server/wls` and *SAMPLES_HOME*`/server/src/examples/iiop/ejb/entity/cppclient` directories).

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 unmarshall it.

Examples of this in the sample *SAMPLES_HOME*`/server/src/examples/iiop/ejb/entity/cppclient` are EJBObjectEnum and Vector. 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 unmarshall the value type manually. See *SAMPLES_HOME*`/server/src/examples/iiop/ejb/enity/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.

# RMI-IIOP Code Examples Package

The `examples.iiop` package is in the `SAMPLES_HOME/server/src/samples/examples/iiop` directory and demonstrates connectivity between numerous clients and applications. The examples demonstrate using EJB's 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 WebLogic Tuxedo Connector, see the `/wlserver6.1/samples/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/docs81b/jndi`.

- *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs81b/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`.