BEA WebLogic Server™

Programming with WebLogic RMI-IIOP

BEA WebLogic Server Version 6.1
Document Date: August 16, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming RMI over IIOP

Part Number	Date	Software Version
	April 29, 2001	BEA Weblogic Server Version 6.1

Contents

About This Document

1. Using WebLogic RMI over IIOP

About This Document

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

This document covers the following topic:

- Chapter 1, "Using WebLogic RMI over IIOP," provides an overview to RMI over IIOP, describes how to develop an RMI over IIOP application, and explains how to configure a WebLogic server. Code segments are provided to illustrate these tasks.

What You Need to Know

This document is intended mainly for application developers who are interested in providing the ability for clients to access Remote Method Invocation (RMI) remote objects using the Internet Inter-ORB Protocol (IIOP). This allows for RMI to Common Object Request Broker Architecture (CORBA) interoperability. It assumes a familiarity with the RMI over IIOP for WebLogic Server 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 or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

How to Print the Document

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

A PDF version of this document is available on the RMI over IIOP for 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 RMI over IIOP for WebLogic Server documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

The following BEA RMI over IIOP for WebLogic Server documents contain information that is relevant to using RMI over IIOP.

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 the BEA RMI over IIOP for WebLogic Server 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 RMI over IIOP for WebLogic Server documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA RMI over IIOP for WebLogic Server 6.0 release.

If you have any questions about this version of BEA RMI over IIOP for WebLogic Server, or if you have problems installing and running BEA RMI over IIOP for WebLogic Server, contact BEA Customer Support through BEA WebSupport at **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	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

Convention	Item

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Using WebLogic RMI over IIOP

The following sections describe features and functionality of RMI over IIOP:

- Introduction
- RMI over IIOP Overview
- RMI-IIOP Programming Models
- RMI over IIOP with an RMI Client
- RMI over IIOP with IDL Client
- RMI-IIOP with Tuxedo and Tuxedo Clients
- Configuring WebLogic Server for RMI-IIOP
- Protocol Compatibility
- Special Considerations
- Code Examples
- Additional Resources

Introduction

WebLogic RMI over IIOP extends the RMI programming model by providing the ability for clients to access RMI remote objects using the Internet Inter-ORB Protocol (IIOP). This exposes RMI remote objects to a new class of client--the Common Object Request Broker Architecture (CORBA) client. 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. The WebLogic Server 6.1 implementation of RMI-IIOP has been greatly revamped and will allow you to do the following:

- Connect with Java RMI clients using the standardized IIOP protocol
- Connect with CORBA/IDL clients, including those written in C++
- Interoperate with a Tuxedo Server
- Connect a variety of clients to EJBs hosted on WebLogic Server!

Within the developer community, there is a strong demand for the ability to access J2EE services from CORBA/IDL clients (hereafter these will be referred to as 'IDL clients'). Since RMI is an enabling technology for EJB, providing RMI over IIOP enhances the ability to support various clients. However, Java and CORBA are based upon 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. As a result, the Objects-by-Value specification was created by the Object Management Group (OMG). 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.

You may also wish to use RMI-IIOP with Java/RMI clients, taking advantage of the standard IIOP protocol. The release of the 1.3.1 JDK has greatly facilitated this capability and WebLogic Server 6.1 can readily be used with RMI-IIOP in a java to java environment.

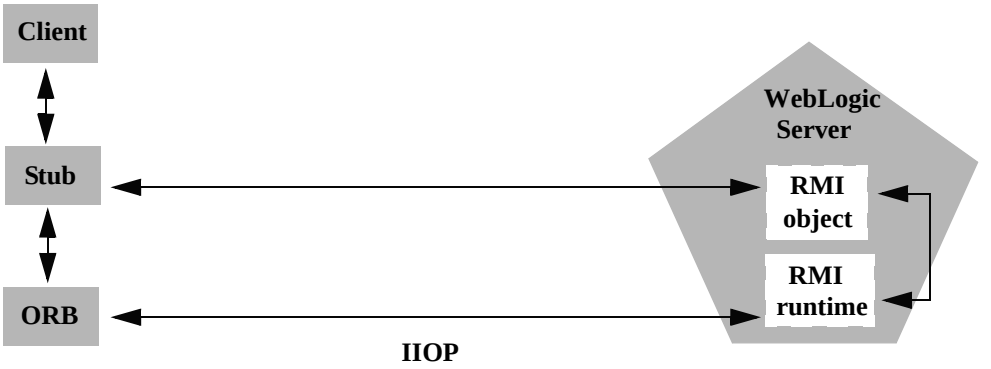
Also, WebLogic Server 6.1 contains an implementation of the Weblogic Tuxedo Connector, an underlying technology which allows you to interoperate with Tuxedo Servers. You can leverage Tuxedo as an ORB, or integrate legacy Tuxedo systems using this feature.

This document describes how to create RMI over IIOP applications for various clients types. How you develop your RMI-IIOP applications will depend on what services and clients you are trying to integrate; read the following sections for further details.

RMI over IIOP Overview

RMI over IIOP is an application of the RMI programming model. In it programmers use JNDI and the RMI type system. For more general information on WebLogic RMI, please refer to *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs61/rmi`; for information on JNDI see *Programming with WebLogic JNDI* at `http://e-docs.bea.com/wls/docs61/jndi`. Both of these technologies are crucial to RMI-IIOP and it highly recommended to become familiar with their general concepts before proceeding to build an RMI-IIOP application.

Figure 1-1 RMI object relationships



RMI-IIOP Programming Models

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. IIOP is a robust protocol that is supported by numerous vendors and is designed to facilitate interoperability of heterogeneous distributed systems. When using RMI-IIOP there are two basic programming models that you may choose to follow when developing applications: 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). They use similar technology, however, the two models are distinctly different approaches to creating a interoperable environment between heterogenous systems. Simply, IIOP can be a transport protocol for distributed applications with interfaces written in either IDL or Java RMI and you must choose between the two. When you program, you must decide to use either IDL or RMI interfaces, you cannot mix them!

RMI over IIOP with RMI Clients combines the features of RMI with the 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.

RMI over IIOP with IDL clients involves an Object Request Broker (ORB) and a compiler that creates an inter-operable language called 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.

For further reference see the following OMG specifications:

- Java Language Mapping to OMG IDL Specification at `http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm`
- CORBA/IIOP 2.4.2 Specification at `http://www.omg.org/cgi-bin/doc?formal/01-02-33`

In addition to these two programming models, there is a third option for those who wish to integrate there BEA Tuxedo clients or services with the WebLogic Server. Using the Weblogic Tuxedo Connector that is included with WebLogic Server eases and strengthens the integration of Tuxedo with WebLogic. The Weblogic Tuxedo Connector uses RMI-IIOP as its underlying mechanism.

Choosing an RMI Programming Model

There are several factors that will define how you will want to create a distributed environment for your applications. Currently there is much confusion surrounding the different models for employing RMI-IIOP. Because these models share many features and standards it is easy to lose clear sight of which model you are following. For sake of clarity, these will now be separated as follows:

- RMI-IIOP with RMI Clients
- RMI-IIOP with IDL Clients
- RMI-IIOP with Tuxedo Clients

The following sections further outline the benefits of each programming model, beginning with (for completeness sake) the basic RMI model:

RMI (Remote-Method-Invocation) 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, and then invoke methods on that object as though it existed locally in the client's virtual machine. This is ideal in a Java to Java paradigm. All RMI-IIOP models are based on RMI, however if you follow a plain RMI model without IIOP then you are not integrating clients written in languages other than Java. For more information see, *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs61/rmi`.

RMI-IIOP with RMI clients is for those oriented towards Java and 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 to 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 specific classes are required, or automatically downloaded in this scenario; this is a good way of having a minimal client distribution. You do not have to use the proprietary t3

protocol used in normal WebLogic RMI. RMI-IIOP with RMI is especially useful for connecting to Enterprise JavaBeans. For further information see, *RMI over IIOP with an RMI Client*

RMI-IIOP with IDL involves following a CORBA programming model and allows interoperability with non-Java clients. If you have existing CORBA applications, you should program according to the RMI-IIOP with 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. For further information see, *RMI over IIOP with IDL Client*.

If you are integrating WebLogic with existing Tuxedo systems you may wish to take advantage of the Weblogic Tuxedo Connector. The Weblogic Tuxedo Connector uses RMI-IIOP to enable clients or services already developed on Tuxedo to be interoperable with WebLogic Server. For further information see, *RMI-IIOP with Tuxedo and Tuxedo Clients*.

Client	Client language	Protocol	Definition	Benefits
RMI	Java	t3	Client that follows the JavaSoft RMI specification	Fast, scalable. Uses optimized WebLogic t3 protocol that improves performance.
RMI over IIOP with RMI Client	Java	IIOP	RMI client that utilizes the CORBA 2.4.2 specification's support for Objects-by-Value. This Java client is developed using the standard RMI/JNDI model.	RMI with Internet-Inter-Orb-Protocol. Use of RMI-IIOP standards. No WebLogic classes required on client.
RMI-IIOP with IDL client	C++, C, Smalltalk, COBOL (any language which has mapping from OMG IDL to that language.)	IIOP	CORBA client that uses a CORBA 2.4.2 ORB. Note: Due to name-space conflicts, Java CORBA clients are not supported by the RMI over IIOP specification.	Interoperability with WebLogic and clients written in C++, COBOL, etc.

Client	Client language	Protocol	Definition	Benefits
RMI-IIOP with Tuxedo Client	C++, C, COBOL (any language which has mapping supplied by Tuxedo from OMG IDL to that language.)	TGIIOP	Tuxedo Server developed with Tuxedo 8.0 or higher	Interoperability between WebLogic Server applications and Tuxedo services

RMI over IIOP with an RMI Client

To develop an RMI over IIOP application, the following steps must be performed:

1. Develop the Remote Interface and Implementation Class and compile with a Java compiler.
2. Generate the IIOP Classes using the `-iiop` option. Note that the IIOP stubs created by the WebLogic RMI compiler are intended to be used with the JDK 1.3 ORB. If you are using another ORB, consult the ORB vendor's documentation to determine whether these stubs are appropriate. Use the -iiopDirectory option to specify a target directory where the IIOP classes will be generated.
3. Develop the RMI Client and compile with a language-specific compiler

See *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs61/rmi` for more general instructions on developing an RMI application.

Develop the Remote Interface and Implementation Class

To develop an RMI object, you must define the object's public methods in an interface that extends `java.rmi.Remote`.Your remote interface may not contain 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 `samples/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;
{
```

With RMI objects, you then implement the interface in a class named `interfaceNameImpl`. 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. The implementation class can be bound into the JNDI tree to be made available to clients. Typically, your implementation class will be configured as a WebLogic startup class and will 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);
```

Once you have developed the remote interface and implementation class, compile them 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.

Generate the IIOP Classes

Run the WebLogic RMI compiler against the implementation class with the `-iiop` option to generate the necessary IIOP stub and skeleton. 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. To run the WebLogic RMI compiler with the `-iiop` option, use the command pattern:

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

In the case of the Pinger example, here the *nameOfImplementationClass* would be `examples.iiop.rmi.server.wls.PingerImpl`. The `-iiopDirectory` option allows you to specify where these iiop classes will be written. The files generated will appear as the *`nameOfInterface_Stub.class`* and the *`nameOfInterface_Skel.class`*. The four files you have now created—the remote interface, the class that implements it, and the stub and skeleton—should be placed in the appropriate directory in the CLASSPATH of the WebLogic Server whose URL you used in the naming scheme of the object's `main()` method.

Develop the RMI Client

RMI clients access remote objects by creating an initial context and performing a lookup on the object. The object is then cast to the appropriate type. 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. To make this example work over IIOP, you must perform the following steps on the client:

- Obtain an initial context.
- Modify the client code to perform the lookup in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

In obtaining an initial context, you have two choices when defining your JNDI context factory:

- `weblogic.jndi.WLInitialContextFactory`
- `com.sun.jndi.cosnaming.CNCtxFactory`

You can use either of these when setting the value for the "`Context.INITIAL_CONTEXT_FACTORY`" property that you supply as a parameter to new `InitialContext()`. If you use the Sun version, you'll have a Sun JNDI client, which in turn uses the Sun RMI-IIOP ORB implementation of J2SE 1.3; this may be important to you if you wish to minimize the use of WebLogic classes on the client. To take full advantage of WebLogic's RMI-IIOP implementation however, it is recommended that you use the `weblogic.jndi.WLInitialContextFactory` method.

There are some items to be aware of when using the Sun JNDI client and the Sun ORB. Their 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:

Listing 1-1 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
* http://edocs.bea.com/wls/docs61/jndi/jndi.html for more
* information

private static Context getInitialContext()
  throws NamingException

{
  return new InitialContext();
}
```

After getting an initial context, `javax.rmi.PortableRemoteObject.narrow()` must be used in any situation where you would normally cast an object to a specific class type. For example, the client code responsible for looking up the EJBean home and casting the result to a `TraderHome` object must be modified to use the `javax.rmi.PortableRemoteObject.narrow()` as shown below:

Listing 1-2 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";
```

To make this client connect over IIOP, you would run client with a command like:

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

In order 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 have to set a security manager in the client with an appropriate java policy file. The following would be an example of a `java.policy` file:

```
grant {

// Allow everything for now

permission java.security.AllPermission;

}
```

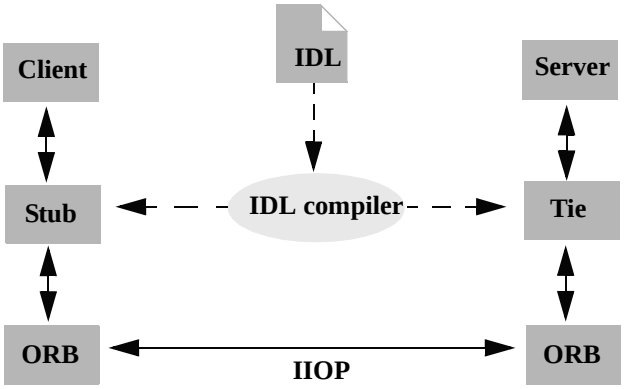
:To set the security manager on the client, do the following:

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

RMI over IIOP with IDL Client

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 which are used by the client and server for obtaining references to remote objects, forwarding requests, and marshalling incoming calls. Note that even with IDL clients it is strongly recommended you begin with the Java remote interface and implementation class as is 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. Begin with the Java remote interface and implementation file and generate the IDL to allow interopability with WebLogic and CORBA clients!

Figure 1-2 IDL Client (Corba object) relationships

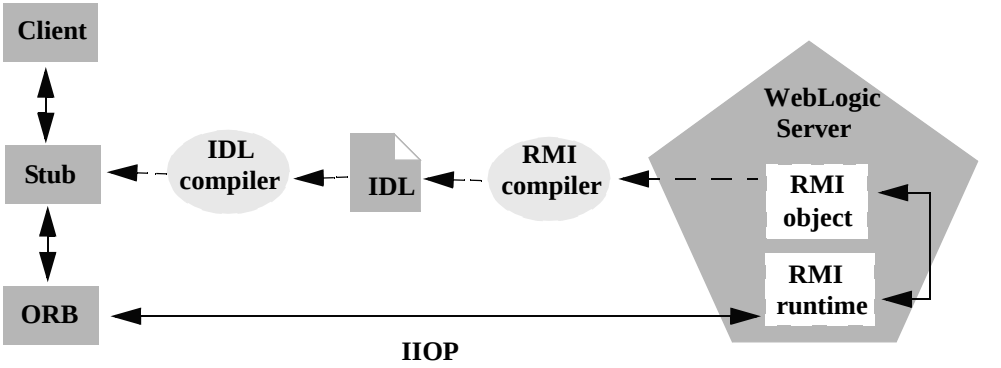


Java IDL Mapping

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, the implementation class is run through the WebLogic RMI compiler or WebLogic EJB compiler with the `-idl` option. This creates an IDL equivalent of the remote interface using the Java-to-IDL mapping specification. This IDL is then compiled 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.

Figure 1-3 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, the client should be developed 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 your RMI over IIOP application, you must consider whether your IDL clients will support Objects-by-Value and design your RMI interface accordingly. In other words, you must limit you RMI interface to pass only primitive data types if your application will support only IDL clients that do not support Objects-by-Value. This will be discussed further in the Value Types section.

Developing an RMI over IIOP Application Using IDL

To develop an RMI over IIOP application with IDL, the following steps must be performed:

1. Develop the Remote Interface and Implementation Class and compile with a Java compiler
2. Generate the IDL File using the WebLogic RMI compiler or WebLogic EJB compiler.

3. Compile the IDL file with an IDL compiler and compile the resulting classes with a language-specific compiler, such as C++.
4. Develop the IDL client and compile with a language-specific compiler

Note: Begin with the remote interface and implementation class! If you are a CORBA programmer you may wish to begin coding with IDL, however this will lead to problems down the road; IDL does not map to Java well. Attempting to generate the Java from the IDL you create(this is sometimes called the 'reverse mapping issue') is not advisable.

Develop the Remote Interface and Implementation Class

To develop an RMI object, you must define the object's public methods in an interface that extends `java.rmi.Remote`.

With RMI objects, you can implement the interface in a class named *`interfaceName`*`Impl`. The implementation class can be bound to the JNDI tree to be made available to clients. Typically, your implementation class will be configured as a WebLogic startup class and will include a `main` method that binds the object into the JNDI tree. The development of the remote interface and implementation class is the same whether you are programming for RMI, IDL, or Tuxedo clients. See the section for Develop the Remote Interface and Implementation Class for more information on this first step. For more information on developing RMI objects, see Using WebLogic RMI.

Special considerations for supporting non-OBV clients

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 we have tested with respect to Objects-by-Value support:

Table 1-1

Vendor	Versions	Objects-by-Value
Borland (formerly Inprise)	VisiBroker 3.3, 3.4	not supported
Borland (formerly Inprise)	VisiBroker 4.x	supported

Table 1-1

Vendor	Versions	Objects-by-Value
Iona	Orbix 2000	supported (we have encountered issues with this implementation)

Generate the IDL File

After developing and compiling the implementation class, you must 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 please 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.
`-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 ejbc)
`-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 will generate the IDL file within sub-directories of the `idlDirectoy` according to the package of the implementation class. For example, the above command will result in a `Hello.idl` file generated in the `/IDL/rmi_iiop` directory. If the `idlDirectory` option is not used, the IDL file will be generated relative to the location of the generated stub and skeleton classes.

Compile the IDL file

Now that you have an IDL file, it can be used 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.

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.

Listing 1-3 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. IDL client applications can then locate an object by asking the CORBA Name Service to look up the name in the JNDI tree of WebLogic Server. 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.

In this case of the example above, you would run the client by using: `Client.exe -ORBInitRef NameService=iioploc://localhost:7001/NameService`.

Note: The naming context can also be obtained by narrowing a CORBA object to the WebLogic IOR. The `host2ior` utility (this utility has been deprecated and is not to be used in production systems) included with WebLogic Server can be used to print the WebLogic Server IOR to the console by running the following command:

```
$  java utils.host2ior hostName port
```

Here, `hostName` is the name of the machine running WebLogic Server and `port` is the port where WebLogic Server is listening for connections.

RMI-IIOP with Tuxedo and 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* at http://e-docs.bea.com/wls/docs61/wtc.html
- For Tuxedo, *CORBA topics* at http://e-docs.bea.com/tuxedo/tux80/interm/corba.htm

BEA WebLogic C++ Client

WebLogic Server 6.1 SP3 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.

Configuring WebLogic Server for RMI-IIOP

Because of a lack of standards for propagating client identity from a CORBA client, the identity of any client connecting over IIOP will default to "guest". The user, as well as a password, can be set in the `config.xml` file to establish a single identity for all clients connecting over IIOP, as shown in the example below:

```
<Server
Name="myserver"
NativeIOEnabled="true"
DefaultIIOPUser="Bob"
```

```
DefaultIIOPPassword="Gumby1234"
ListenPort="7001">
```

There is also a `IIOPEnabled` attribute which can be set in the `config.xml`. The default value `"true"` and set this to `"false"` only if you wish 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 6.1 supports RMI-IIOP `corbaname` and `corbaloc` JNDI references. Please refer to the CORBA/IIOP 2.4.2 Specification. So, for instance, the following could be added to your `ejb-jar.xml`:

```
<ejb-reference-description>
<ejb-ref-name>WLS</ejb-ref-name>
<jndi-name>corbaname:iiop:1.2@localhost:7001#ejb/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 6.1 contains an implementation of GIOP 1.2. The GIOP specifies formats for messages that are exchanged between inter-operating ORBs. This allows interoperatability with many other ORB's and application servers. The GIOP version can be controlled by the version number in a `corbaname` or `corbaloc` reference.

Protocol Compatibility

Interoperability between WebLogic Server 6.x and WebLogic Server 8.1 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 1-2 WebLogic Server-to-Server Interoperability

To Server From 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 HTTP Web Services[13]	RMI/T3 RMI/IIOP HTTP Web Services
Sun JDK ORB client[14]	RMI/IIOP[15]	RMI/IIOP[16]	RMI/IIOP[17]	RMI/IIOP[18]

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 transaction propagation. Known problems with exception marshalling
13. Must use portable client stubs generated from the “To Server” version
14. This option involves calling directly into the JDK ORB from within application hosted on WebLogic Server.
15. JDK 1.3.x only. No clustering. No transaction propagation
16. JDK 1.3.x only. No clustering. No transaction propagation
17. JDK 1.3.x or 1.4.1. No clustering. No transaction propagation
18. 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 1-3 Client-to-Server Interoperability

To Server From Client (stand-alone)	WebLogic Server 6.0	WebLogic Server 6.1	WebLogic Server 7.0	WebLogic Server 8.1
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]

To Server From Client (stand-alone)	WebLogic Server 6.0	WebLogic Server 6.1	WebLogic Server 7.0	WebLogic Server 8.1
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]
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
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 ant 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

Special Considerations

The following sections provide additional information about special considerations you may need to consider when designing applications that use RMI-IIOP:

- RMI-IIOP and the RMI Object Lifecycle
- Limiations on Using RMI-IIOP on the Server
- Limitations on Using RMI-IIOP on the Client
- Java and the IDL Client Model
- Using EJBs with RMI-IIOP
- RMI over IIOP with SSL
- Accessing WebLogic Server Objects from a CORBA Client Through Delegation

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.

Limiations on Using RMI-IIOP on the Server

If you are using RMI-IIOP on the server, note the following limitations:

- Clustering support for RMI objects that run over the IIOP protocol is limited to server-side objects.
- Clustered URLs are not supported.
- Load balancing and failover is suppported for clustered objects running over IIOP only if they run within the WebLogic Server runtime environment.

Limitations on Using RMI-IIOP on the Client

JDK 1.3 (and various versions) are not RMI-IIOP comformant. If you are using RMI-IIOP on the client, note the following about the JDK:

- It does not support client-demarcated transactions.
- It sends GIOP 1.0 messages and GIOP 1.1 profiles in IORs.
- It does not support the necessary pieces for EJB 2.0 interoperability (GIOP 1.2, codeset negotiation, UTF-16).
- It has some bugs in its treatment of mangled method names.
- It does not correctly unmarshal unchecked exceptions.
- It has some subtle bugs relating to the encoding of valuetypes.
- It does not support clustering (failover and load balancing).

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

Java and the IDL Client Model

WebLogic strongly recommends developing Java clients with the RMI client model if you are going to use RMI-IIOP. If you are developing a Java IDL Client you will encounter many difficulties. Naming conflicts and keeping the server-side and client-side classes separate--as well as classpath problems--are some of the more obvious problems. Since 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.

Using EJBs with RMI-IIOP

When Enterprise JavaBeans are implemented using RMI over IIOP for EJB interoperability in heterogeneous server environments, the standard mapping of the EJB architecture to CORBA enables the following:

- A Java RMI client using an ORB can access enterprise beans residing on a WebLogic Server over IIOP.
- A non-Java platform CORBA client can access any enterprise bean object on WebLogic Server.

WebLogic RMI over IIOP is the framework for which EJBs can connect to IDL clients. Currently however, a standard for passing user identity--required to implement EJB-to-IDL-- does not exist and the requirement for transaction propagation from the client is in question. While RMI over IIOP does allow CORBA/IDL clients to access EJBeans, the following services will not be available:

- EJB transaction services
- EJB security services

When deriving the mapping from a WebLogic Server application to a CORBA client, the sources of the mapping information are the EJB classes as defined in the Java source files. WebLogic Server provides the `weblogic.ejbc` utility for generating required IDL files. These files represent the CORBA view into the state and behavior of the target EJB. The `weblogic.ejbc` utility will allow the following:

- 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.ejbc` utility supports a number of command qualifiers. See the chart at, Generate the IDL File.

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 following section for more information). Generated IDL files are placed in the `idlSources` directory. Note that the java to IDL process is full of pitfalls and please reference the Java Language Mapping to OMG IDL specification at `http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm`.

The following is an example (from *`WL_HOME`*`/examples/iiop/ejb/entity/server/wls` in your distribution) of how to generate the IDL from a bean you have already created:

```
> java weblogic.ejbc -compiler javac -keepgenerated
-idl -idlDirectory idlSources
-iiop 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 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.ejbc`, 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.

Value Types

When you wish to pass objects by value rather than reference you will need to use Value Types (see chapter five of the CORBA/IIOP 2.4.2 Specification for further information) Value types need to be implemented on each platform on which they are defined or referenced. The following section deal with the difficulties of passing complex valuetypes, referencing the particular case of a C++ client accessing an Entity bean on WebLogic Server (see the `examples/iiop/ejb/entity/server/wls` and `examples/iiop/ejb/entity/cppclient` directories).

One of the problems encountered by Java programmers is the use of derived datatypes that are not usually visible. For instance when accessing an EJB finder the Java programmer will see a Collection or Enumeration, but doesn't pay attention to the underlying implementation because the JDK run-time will have it classloaded over the network. However the C++, CORBA programmer has to know the type that comes across the wire so that he can register a valuetype factory for it so that the ORB can unmarshall it.

Examples of this in the sample `/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. Please review the `/iiop/ejb/entity/cppclient` example to see how to register a valuetype factory.

Java types that are serializable but that define `writeObject()` are mapped to custom valuetypes in IDL. You will have to write C++ code to hand unmarshall the valuetype. See `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 directs 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\rmi_iiop\ejb\rmi_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 over IIOP with SSL

The SSL protocol can be used to protect IIOP connections to RMI or EJB remote objects. The SSL protocol secures connections through authentication and encrypts the data exchanged between objects. You can use RMI over IIOP over SSL in WebLogic Server in the following ways:

- With a CORBA/IDL client Object Request Broker (ORB)
- With a Java client

In either case, you need to configure WebLogic Server to use the SSL protocol. For more information, see Configuring the SSL Protocol.

To use RMI over IIOP over SSL with a CORBA/IDL client ORB, do the following:

1. Configure the CORBA client ORB to use the SSL protocol. Refer to the product documentation for your client ORB for information about configuring the SSL protocol.
2. Use the `host2ior` utility to print the WebLogic Server IOR to the console. The `host2ior` utility prints two versions of the IOR, one for SSL connections and one for non-SSL connections.
3. Use the SSL IOR when obtaining the initial reference to the `CosNaming` service that accesses the WebLogic Server JNDI tree.

To use RMI over IIOP over SSL with a Java client, do the following:

1. If you want to use callbacks, obtain a private key and digital certificate for the Java client.
2. Run the `ejbc` compiler with the `-d` option.
3. Use the command options below when starting the RMI client. Note that you must specify a machine name, your regular port and then the SSL port. Also, you must use the `weblogic.corba.orb.ssl.ORB` class which wraps around the Orb's own class and fixes problem with the JDK handling secure connections:

```
java -Dweblogic.security.SSL.ignoreHostnameVerification=true \
-Dweblogic.SSL.ListenPorts=localhost:7701:7702 \
-Dorg.omg.CORBA.ORBClass=weblogic.corba.orb.ssl.ORB \
weblogic.rmiiiop.HelloJDKClient iiop://localhost:7702
*
```

```
* or to use cert chains for Server to Client connections:
*

*java -Dweblogic.corba.orb.ssl.certs=myserver/democert.pem
-Dweblogic.corba.orb.ssl.key=myserver/demokey.pem
-Dweblogic.security.SSL.ignoreHostnameVerification=true
-Dweblogic.corba.orb.ssl.ListenPorts=localhost:7701:7702
-Dorg.omg.CORBA.ORBClass=weblogic.corba.orb.ssl.ORB
-Djava.security.manager -Djava.security.policy==java.policy -ms32m
-mx32m weblogic.rmiiop.HelloJDKClient port=7702
```

```
-Dssl.certs=directory location of digital certificate for Java
client
-Dssl.key=directory location of private key for Java client
```

The Java client needs to have the classes that WebLogic Server uses for the SSL protocol included in its `CLASSPATH`.

For incoming connections (from WebLogic Server to the Java client for the purpose of callbacks), you need to specify a digital certificate and private key for the Java client on the command line. Use the `ssl.certs` and `ssl.key` command-line options to provide this information. Using Client Certificates

Once you have set the client ORB to support SSL, you can enforce an additional level of security by using client certificates with RMI over IIOP and SSL. The behavior will differ depending on whether you choose to enforce client certificates.

The client ORB must be aware of WebLogic Server's trusted certificate authenticator and WebLogic Server must be aware of the ORB's trusted certificate authenticator. To make WebLogic Server aware of the ORB's certificate authenticator copy the client ORB's trusted certificate authenticator to WebLogic Server.

1. Use java `utils.der2pem` to convert the certificate authenticator.
2. Copy the `ca.pem` file to a new `ca_new.pem` file.
3. Add the client ORB's trusted `ca.pem` file to the end of the new `ca_new.pem` file.
4. In the Console, change the Trusted CAFile Name to `ca_new.pem`.

   The certificate chain file will still be `ca.pem`.

Note: Refer to the ORB's product documentation to see how to make the client ORB aware of WebLogic Server's trusted certificate authenticator.

Implement the `weblogic.security.acl.CertAuthenticator` interface and register the class in the Console. See `examples.security.cert`, in your WebLogic Server distribution for a sample of how this is handled.

Using the Administration Console, set the client certificate enforced option:

1. Click the Server tab and in the right pane, choose the desired server.
2. Click the SSL tab and select the Client Certificate Enforced box.
3. Click Apply.

With RMI over IIOP and SSL, you can expect the following behavior:

If client certificates are enforced:

- The client ORB invokes on the IOR using SSL.
- On the server side, the Certificate Authenticator class is called to determine if the user is authorized.

  To configure the certificate authenticator, choose on the SSL tab and specify the certificate authenticator to be used to determine the validity of the certificate. The certificate authenticator class accesses the certificate and uses the fields on the certificate to determine the user.

  The *`WL_HOME`*`/samples/examples/security/cert/` example contains a very simple certificate authenticator class that maps the email address from a certificate to a user in the realm.
- If the certificate authenticator is not configured or the certificate authenticator returns `null`, then a no permission exception is returned to the client and the method is not executed.

If client certificates are not enforced:

- The client ORB invokes on the IOR using SSL.
- On the server side, the invoke occurs on the default IIOP user.

  To set this user option on the SSL tab, click the Protocols tab and check the Default IIOP Users box.

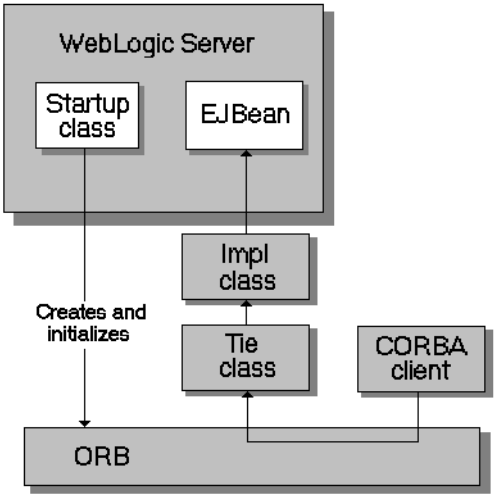
Accessing WebLogic Server Objects from a CORBA Client Through Delegation

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. This document provides an overview of how this is done.

Overview

There are a number of objects that must work together to be able to delegate CORBA calls to an object hosted by WebLogic Server. First, you must have an ORB that is co-located with the JVM that is running WebLogic Server. To accomplish this, you can create a startup class that creates and initializes an ORB. You also need an object that will accept incoming messages from the ORB. To create this object, you must create an IDL (Interface Definition Language). Compiling the IDL will result in 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 throughout the JNDI tree and delegate the appropriate calls to the CORBA object.



Delegate architecture

Code Example

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

For more information on how to implement a startup class, see *Starting and Stopping WebLogic Servers* at `http://e-docs.bea.com/wls/docs61/adminguide/startstop.html`.

```
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();

    }

  }

}
```

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 `/wlserver6.1/samples/examples/wtc` directory.

The following table provides information on the RMI-IIOP examples provided for WebLogic Server 6.1.

Figure 1-4 WebLogic Server 6.1 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 4.1	GIOP 1.0 protocol. Users must add the `DefaultGIOPMinorVersion` attribute and set its value to "1" in the Server MBean of the `config.xml file`.
`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.

Example	ORB/Protocol	Requirements
`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 4.1	GIOP 1.0 protocol. Users must add the `DefaultGIOPMinorVersion` attribute and set its value to "1" in the Server MBean of the `config.xml file`.
`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 1.3.1	JDK 1.3.1 requires a security policy file to access server.
`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 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	

Example	ORB/Protocol	Requirements
`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.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 4.1	GIOP 1.0 protocol. Users must add the `DefaultGIOPMinorVersion` attribute and set its value to "1" in the Server MBean of the `config.xml file`.
`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	
`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 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	Requirements
`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.

Additional Resources

WebLogic RMI over 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/docs61/jndi`.
- *Using WebLogic RMI* at `http://e-docs.bea.com/wls/docs61/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`.