BEA WebLogic Server®

Programming Stand-alone Clients

Version 9.0 BETA
Revised: December 15, 2004

Contents

Introduction and Roadmap

This section describes the contents and organization of this guide—*Programming Stand-alone Clients*.

- “Document Scope and Audience” on page 1-1
- “Guide to this Document” on page 1-2
- “Related Documentation” on page 1-2
- “Samples and Tutorials” on page 1-2
- “New and Changed Features in This Release” on page 1-3

Document Scope and Audience

This document is a resource for software developers who want to develop stand-alone clients that inteoperate with WebLogic Server®.

The topics in this document are relevant during the design and development phases of a software project. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

It is assumed that the reader is familiar with J2EE and JMS concepts. This document emphasizes the value-added features provided by WebLogic Server and key information about how to use WebLogic Server features and facilities when developing stand-alone clients.

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.
- Chapter 2, "Overview of a Client Application," describes basic client- server functionality.
- Chapter 3, "Using RMI over IIOP Programming Models to Develop Applications,"provides information on o use various programming models to develop RMI-IIOP applications.
- Chapter 4, "WebLogic JMS Thin Client,"provides information on how to develop and deploy a WebLogic JMS thin client.

Related Documentation

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Developing WebLogic Server Applications* is a guide to developing WebLogic Server applications.
- *Deploying WebLogic Server Applications* is the primary source of information about deploying WebLogic Server applications.

Samples and Tutorials

In addition to this document, BEA Systems provides a variety of code samples and tutorials for JMS developers. The examples and tutorials illustrate WebLogic Server JMS in action, and provide practical instructions on how to perform key JMS development tasks.

BEA recommends that you run some or all of the JMS examples before developing your own EJBs.

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample J2EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and J2EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

MedRec includes a service tier comprised primarily of Enterprise Java Beans (EJBs) that work together to process requests from web applications, web services, and workflow applications, and future client applications. The application includes message-driven, stateless session, stateful session, and entity EJBs.

Examples in the WebLogic Server Distribution

WebLogic Server 9.0 optionally installs API code examples in `WL_HOME\samples\server\examples\src\examples`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server 9.0 Start menu.

New and Changed Features in This Release

For release-specific information, see these sections in *WebLogic Server 9.0 Release Notes*:

- "WebLogic Server 9.0 Features and Changes" lists new, changed, and deprecate features.
- "WebLogic Server 9.0 Known Issues" lists known problems by service pack, for all WebLogic Server APIs, including EJB.

For more release-specific information about the hardware and software configurations supported by BEA for this release of WebLogic Server, see WebLogic Platform Supported Configurations.

Overview of a Client Application

Java clients that access WebLogic Server application modules range from simple command line utilities that use standard I/O to highly interactive GUI applications built using the Java Swing/AWT classes. Java clients access WebLogic Server modules indirectly through HTTP requests or RMI requests. The modules execute requests in WebLogic Server, not in the client.

In previous versions of WebLogic Server, a Java client required the full WebLogic Server JAR on the client machine. Releases of WebLogic Server 8.1and higher support a true J2EE application client, referred to as the *thin client*. A small footprint standard JAR and a JMS JAR—`wlclient.jar` and `wljmsclient.jar` respectively—are provided in the `/server/lib` subdirectory of the WebLogic Server installation directory. Each JAR file is about 400 KB.

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 JavaBeans running in the business tier, and may, as appropriate communicate through HTTP with servlets running in the Web tier. Although a J2EE application client is a Java application, it differs from a stand-alone Java application client because it is a J2EE module, hence it offers the advantages of portability to other J2EE-compliant servers, and can access J2EE services. For more information about the thin client, see *"Developing a J2EE Application Client (Thin Client)" on page 3-15*.

More information to follow.

CHAPTER 3

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

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

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

Client Types and Features

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

Table 3-1 WebLogic Server Client Types and Features

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

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

ORB Implementation

WebLogic Server provides its own ORB implementation in place of the ORB shipped with J2SE 1.4. This ORB is instantiated by default when programs call `ORB.init()`, or when `"java:comp/ORB"` is looked up in JNDI.

Using a Foreign ORB

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

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

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

Using a Foreign RMI-IIOP Implementation

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

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

You will get the following errors at server startup:

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

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

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

The J2SE defaults for these properties are:

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

Developing a 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/docs90/rmi.`

Developing a J2SE Client

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

When to Use a J2SE Client

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

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

Procedure for Developing J2SE Client

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

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

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

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

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

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

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

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

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

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

6. Obtain an initial context.

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

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

Note: The Sun JNDI client supports the capability to read remote object references from the namespace, but not generic Java serialized objects. This means that you can read items such as EJBHomes out of the namespace but not DataSource objects. There is also no support for client-initiated transactions (the JTA API) in this configuration, and no support for security. In the stateless session bean RMI Client example, the client obtains an initial context as is done below:

Obtaining an InitialContext:

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

   */

  private Context getInitialContext() throws NamingException {

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

/**

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

private static Context getInitialContext()
  throws NamingException

{
  return new InitialContext();
}
```

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

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

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

`Performing a lookup:`

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

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

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

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

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

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

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

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

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

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

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

9. Set the security manager on the client:

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

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

```
grant {

// Allow everything for now

permission java.security.AllPermission;

}
```

Developing a J2EE Application Client (Thin Client)

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

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

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

The thin client is based upon the RMI-IIOP protocol stack and leverages features of J2SE 1.4 It also requires the support of the JDK ORB. The basics of making RMI requests are handled by the JDK, enabling a significantly smaller client. Client-side development is performed using standard J2EE APIs, rather than WebLogic Server APIs.

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

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

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

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

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

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

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

Procedure for Developing J2EE Application Client (Thin Client)

To develop a J2EE Application Client:

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

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

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

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

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

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

   }
   ```

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

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

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

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

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

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

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

6. Obtain an initial context.

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

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

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

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

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

   Performing a lookup:

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

/**
 * Lookup the EJBs home in the JNDI tree
 */
private TraderHome lookupHome()
  throws NamingException
```

```
{
  // Lookup the beans home using JNDI
  Context ctx = getInitialContext();

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

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

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

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

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

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

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

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

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

Developing Security-Aware Clients

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

Developing Clients that Use JAAS

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

Thin-client Restrictions for JASS

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

- UsernamePasswordLoginModule
- Security.runAs

Developing Clients that use SSL

BEA WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between WebLogic Server clients and servers, Java clients, Web browsers, and other servers. You can find more information on how to implement SSL in Using SSL Authentication in Java Clients.

Thin-client Restrictions for SSL

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

```
.
.
.
// Get a KeyManagerFactory for KeyManagers
System.out.println("Retrieving KeyManagerFactory & initializing");
    KeyManagerFactory kmf =
    KeyManagerFactory.getInstance("SunX509","SunJSSE");
    kmf.init(ks,keyStorePassword);
```

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

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

Security Code Examples

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

Developing a WLS-IIOP Client

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

To support WLS-IIOP clients, you must:

- have the full `weblogic.jar` (located in *`WL_HOME`*`/server/lib`) in the client’s CLASSPATH.
- use `weblogic.jndi.WLInitialContextFactory` when defining your JNDI context factory. Use this class when setting the value for the "`Context.INITIAL_CONTEXT_FACTORY`" property that you supply as a parameter to new `InitialContext()`.

Otherwise, the procedure for developing a WLS-IIOP Client is the same as the procedure described in "Developing a J2SE Client" on page 3-11.

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

Developing a CORBA/IDL Client

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

Guidelines for Developing a CORBA/IDL Client

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

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

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

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

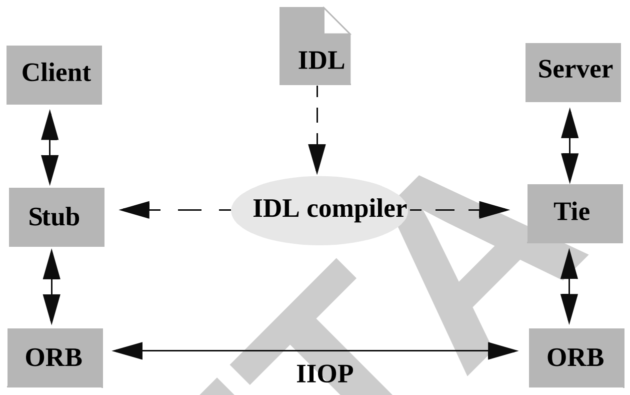
– CORBA/IIOP 2.4.2 Specification at http://www.omg.org/cgi-bin/doc?formal/01-02-33

Working with CORBA/IDL Clients

In CORBA, interfaces to remote objects are described in a platform-neutral interface definition language (IDL). To map the IDL to a specific language, the IDL is compiled with an IDL compiler. The IDL compiler generates a number of classes such as stubs and skeletons that the client and server use to obtain references to remote objects, forward requests, and marshall incoming calls. Even with IDL clients it is strongly recommended that you begin programming with the Java remote interface and implementation class, then generate the IDL to allow interoperability with WebLogic and CORBA clients, as illustrated in the following sections. Writing code in IDL that can be then reverse-mapped to create Java code is a difficult and bug-filled enterprise and WebLogic does not recommend doing this.

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

Figure 3-1 IDL Client (Corba object) relationships



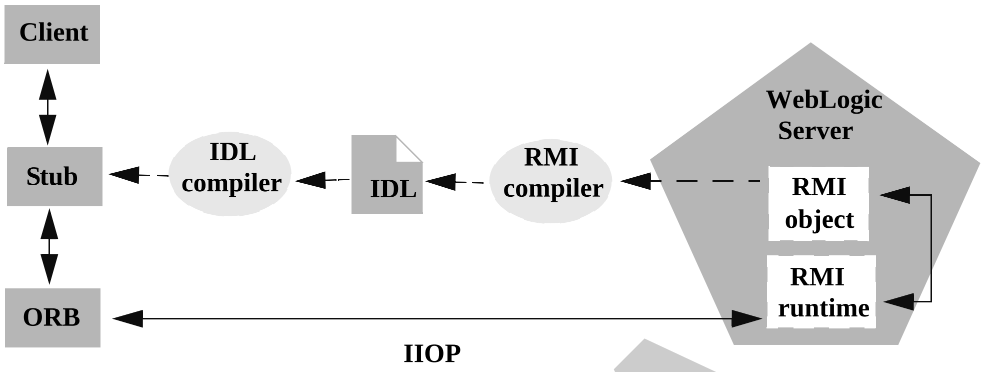
Java to IDL Mapping

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 3-2 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 3-2 ORBs Tested with Respect to Objects-by-Value Support

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

For more information on Objects-by-Value, see "Limitations of Passing Objects by Value" in *Programming WebLogic RMI over IIOP*.

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 3-11.
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/technology/documents/formal/java_language_mapping_to_omg_idl.htm.

   The following compiler options are specific to RMI over IIOP:

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

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

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

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

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

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

4. Develop the IDL client.

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

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

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

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

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

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

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

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

}
```

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

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

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

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

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

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

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

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

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

When to Use a WebLogic C++ Client

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

- To simplify your development process by avoiding third-party products
- To provide a client-side solution that allows you to develop or modify existing C++ clients

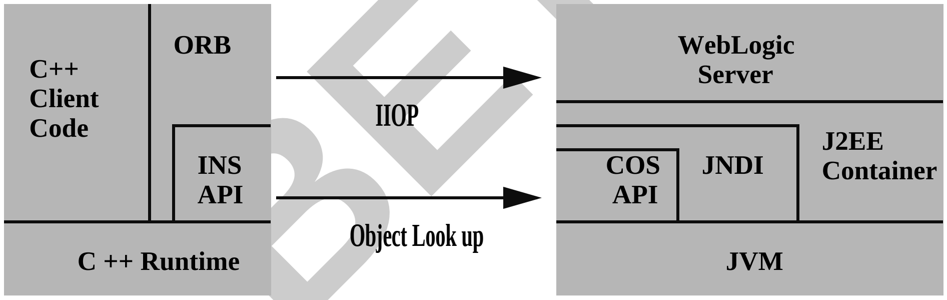
Although the Tuxedo C++ Client ORB is packaged with Tuxedo 8.1 and higher, you do not need a Tuxedo license to develop WebLogic C++ clients. You can obtain a trial development copy of Tuxedo from the BEA Download Center.

How the WebLogic C++ Client works

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

- The WebLogic C++ client code requests a WebLogic Server service.
  - The Tuxedo ORB generates an IIOP request.
  - The ORB object is initally instantiated and supports Object-by-Value data types.
- The Client uses the CORBA Interoperable Name Service (INS) to look up the EJB object bound to JNDI naming service. For more information on how to use the Interoperable Naming Service to get object references to initial objects such as NameService, see Interoperable Naming Service Bootstrapping Mechanism.

Figure 3-3 WebLogic C++ Client to WebLogic Server Interoperability



Developing WebLogic C++ Clients

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

1. Use the `ejbc` compiler with the `-idl` option to compile the EJB that your C++ client will interoperate with. This will generate an IDL script for the EJB.
2. Use the C++ IDL compiler to compile the IDL script and generate the CORBA client stubs, server skeletons, and header files. For information on the use of the C++ IDL Compiler, see OMG IDL Syntax and the C++ IDL Compiler.
3. Discard the server skeletons as the EJB represents the server side implementation.

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

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

WebLogic C++ Client Limitations

The WebLogic C++ client has the following limitations:

- Provides security through the WebLogic Server Security service.
- Provides only server-side transaction demarcation.

WebLogic C++ Client Code Samples

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

RMI-IIOP Applications Using WebLogic Tuxedo Connector

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

When to Use WebLogic Tuxedo Connector

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

How the WebLogic Tuxedo Connector Works

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

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

- The WebLogic Tuxedo Connector Guide at http://e-docs.bea.com/wls/docs90/wtc.html
- For Tuxedo, CORBA topics at http://e-docs.bea.com/tuxedo/tux80/interm/corba.htm

WebLogic Tuxedo Connector Code Samples

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

Using the CORBA API

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

- Support of out and inout parameters
- Support for a call to a CORBA service from WebLogic Server using transactions and security.
- Support for a WebLogic ORB hosted in JNDI rather than an instance of the JDK ORB used in previous releases.

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

- "Supporting Outbound CORBA Calls" on page 3-30
- "Using the WebLogic ORB Hosted in JNDI" on page 3-31
- "Supporting Inbound CORBA Calls" on page 3-32
- "Limitation When Using the CORBA API" on page 3-33

Supporting Outbound CORBA Calls

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

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

3. Build EJB(s) including the generated stubs in the jar.
4. Use the WebLogic ORB hosted in JNDI to reference the external service.

Using the WebLogic ORB Hosted in JNDI

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

ORB from JNDI

```
.

.

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

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

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

.

.

.
```

Direct ORB creation

```
.

.

.

ORB orb = ORB.init();

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

.

.

.
```

Using JNDI

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

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

Supporting Inbound CORBA Calls

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

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


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

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

Limitation When Using the CORBA API

CORBA Object Type support has the following limitations:

- It should not be used to make calls from one WebLogic Server instance to another WebLogic Server instance.
- It does not support clustering. If a clustered object reference is detected, WebLogic Server will use internal RMI-IIOP support to make the call. Any out or inout parameters will not be supported.
- CORBA services created by `ORB.connect()` result in a second object hosted inside the server. It is important that you use `ORB.disconnect()` to remove the object when it is no longer needed.

Using EJBs with RMI-IIOP

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

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

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

- Place the EJB classes, interfaces, and deployment descriptor files into a JAR file.
- Generate WebLogic Server container classes for the EJBs.

- Run each EJB container class through the RMI compiler to create stubs and skeletons.
- Generate a directory tree of CORBA IDL files describing the CORBA interface to these classes.

The `weblogic.appc` utility supports a number of command qualifiers. See "Procedure for Developing a CORBA/IDL Client" on page 3-25.

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 see "Limitations of WebLogic RMI over IIOP" in *Programming WebLogic RMI over IIOP* 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/examples/src/examples/iiop` directory.

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.

WebLogic JMS Thin Client

The following sections describe how to deploy and use the WebLogic JMS thin client:

Overview of the JMS Thin Client

While the size of the full WebLogic JAR may not be a problem when you run server-side applications, it does cause a very large footprint for enterprise-level client-server applications that may be running thousands of clients. Having to deploy the full 20+ MB `weblogic.jar` file along with a client application can significantly increase the size of the deployed application, possibly making it too big to be practical, as is the case with a Java applet-based client program.

At around 400 KB, the JMS thin application client (`wljmsclient.jar`) file provides a much smaller client footprint than the full WebLogic JAR, by using a client-side library that contains only the set of supporting files required by client-side programs. The JMS thin client also requires that you use the standard WebLogic thin application client (`wlclient.jar`), around 300 KB, which contains the base client support for clustering, security, and transactions.

The WebLogic thin application clients are based upon the RMI-IIOP protocol stack available in the JRE. RMI requests are handled by the JRE, enabling a significantly smaller client. Client-side development is performed using standard J2EE APIs, rather than WebLogic Server APIs.

For more information on developing WebLogic Server thin client applications, see "Developing a J2EE Application Client (Thin Client)" on page 3-15.

JMS Thin Client Functionality

Although much smaller in size than the full WebLogic JAR, the JMS thin client and WebLogic Server thin clients provide the following functionality to client applications and applets:

- Full WebLogic JMS functionality—both standard JMS and WebLogic extensions—except for client-side XML selection for multicast sessions and the JMSHelper class methods
- EJB (Enterprise Java Bean) access
- JNDI access
- RMI access (indirectly used by JMS)
- SSL access (using JSSE in the JRE)
- Transaction capability
- Clustering capability
- HTTP/HTTPS tunneling
- Fully internationalized

Limitations of Using the JMS Thin Client

The following limitations apply to the JMS thin client:

- It does not provide the JDBC or JMX functionality of the normal `weblogic.jar` file.
- It does not interoperate with earlier versions of WebLogic Server (release 7.0 or earlier).
- It is only supported by the JDK ORB.

Deploying the JMS Thin Client

The JMS thin client and WebLogic thin client JARs (`wljmsclient.jar` and `wlclient.jar`, respectively) are located in the *`WL_HOME`*`\server\lib` subdirectory of the WebLogic Server installation directory, where *`WL_HOME`* is the top-level installation directory for the entire WebLogic Platform (for example, `c:\bea\weblogic81\server\lib`).

Deployment of the JMS thin client depends on the following requirements:

- The JMS thin client requires that you use the standard WebLogic thin client, which contains the base client support for clustering, security, and transactions. Therefore, the `wljmsclient.jar` and the `wlclient.jar` must be installed somewhere on the client's file system. However, `wljmsclient.jar` has a reference to `wlclient.jar` so it is only necessary to put one or the other Jar in the client's `CLASSPATH`.
- The JMS thin client requires using the RMI over IIOP standard for communicating between client and server.
  - URLs using `t3` or `t3s` will transparently use `iiop` or `iiops`
  - URLs using `http` or `https` will transparently use `iiop` tunneling.
- To facilitate the use of IIOP, always specify a valid IP address or DNS name for the Listen Address attribute in the configuration file (`config.xml`) to listen for connections.

  Note: The Listen Address default value of `null` allows it to "listen on all configured network interfaces". However, this feature only works with the T3 protocol. If you need to configure multiple listen addresses for use with the IIOP protocol, then use the Network Channel feature, as described in "Configuring Network Resources" in *Designing and Configuring WebLogic Server Environments*.
- Each client must have the JRE 1.4.*n* or higher installed.
- Applications must adhere to J2EE programming guidelines, in particular the use of `PortableRemoteObject.narrow()` rather than using casts.

For more information on developing WebLogic Server thin client applications, see "Developing a J2EE Application Client (Thin Client)" on page 3-15.