bea

BEAWebLogic Server®

Programming WebLogic RMI

Version 10.0
Document Revised: March 30, 2007

Contents

4. Using the WebLogic RMI Compiler

5. Using WebLogic RMI with T3 Protocol

6. How to Implement WebLogic RMI

7. Using RMI over IIOP

8. Configuring WebLogic Server for RMI-IIOP

9. Best Practices for Application Design

A. CORBA Support for WebLogic Server

Introduction and Roadmap

This section describes the contents and organization of this guide—*Programming WebLogic RMI*.

- "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 1-3

- "New and Changed Features in This Release" on page 1-4

Document Scope and Audience

This document is written for application developers who want to build e-commerce applications using Remote Method Invocation (RMI) and Internet Interop-Orb-Protocol (IIOP) features. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language. This document emphasizes the value-added features provided by WebLogic Server® and key information about how to use WebLogic Server features when developing applications with RMI.

Guide to this Document

This document describes the BEA WebLogic Server RMI implementation of the JavaSoft^{TM} Remote Method Invocation (RMI) specification from Sun Microsystems. The BEA implementation is known as WebLogic RMI.

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.

- Chapter 2, "Understanding WebLogic RMI," is an overview of WebLogic RMI features and its architecture.

- Chapter 3, "WebLogic RMI Features," describes the features that you use to program RMI for WebLogic Server.

- Chapter 4, "Using the WebLogic RMI Compiler," provides information on the WebLogic RMI compiler.

- Chapter 6, "How to Implement WebLogic RMI," provides a simple step by step example of how to implement WebLogic RMI.

- Chapter 7, "Using RMI over IIOP," defines RMI over IIOP and provides general information about the WebLogic Server RMI-IIOP implementation.

- Chapter 8, "Configuring WebLogic Server for RMI-IIOP," describes concepts, issues, and procedures related to using WebLogic Server to support RMI-IIOP applications.

- Chapter 9, "Best Practices for Application Design," describes recommended design patterns when developing RMI and RMI over IIOP applications.

- Appendix A, "CORBA Support for WebLogic Server," provides information on CORBA support for WebLogic Server.

Related Documentation

For information on topics related to WebLogic RMI, see the following documents:

- *Java(TM) Remote Method Invocation (RMI)* is a link to basic Sun MicroSystems tutorials on Remote Method Invocation.

- *Developing Applications with WebLogic Server* is a guide to developing WebLogic Server applications.

- *Programming WebLogic JNDI* is a guide using the WebLogic Java Naming and Directory Interface.

- *Programming Stand-alone Clients* is a guide to developing common stand alone clients that interoperate with WebLogic Server.

- CORBA Technology and the Java Platform provides an overview of CORBA and Java platform.

- Java IDL Technology contains information using standard IDL (Object Management Group Interface Definition Language) and IIOP.

- omg.org is the Object Management Group homepage.

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

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

Samples and Tutorials

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

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

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample Java Platform, Enterprise Edition (Java EE) 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 Java EE 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 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 Start menu.

Additional Examples Available for Download

Additional API examples for download at http://dev2dev.bea.com/. These examples are distributed as `.zip` files that you can unzip into an existing WebLogic Server samples directory structure. You build and run the downloadable examples in the same manner as you would an installed WebLogic Server example. See the download pages of individual examples for more information.

New and Changed Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in release, see "What's New in WebLogic Server 10.0" in *Release Notes*.

Understanding WebLogic RMI

The following sections introduce and describe the features of WebLogic RMI.

-
-

What is WebLogic RMI?

Remote Method Invocation (RMI) is the standard for distributed object computing in Java. RMI enables an application to obtain a reference to an object that exists elsewhere in the network, and then invoke methods on that object as though it existed locally in the client's virtual machine. RMI specifies how distributed Java applications should operate over multiple Java virtual machines.

This document contains information about using WebLogic RMI, but it is not a beginner's tutorial on remote objects or writing distributed applications. If you are just beginning to learn about RMI, visit the JavaSoft Web site and take the RMI tutorial.

Features of WebLogic RMI

The following table highlights important features of WebLogic implementation of RMI:

Table 1-1 WebLogic RMI Features

Feature	WebLogic RMI
Overall performance	Enhanced by WebLogic RMI integration into the WebLogic Server framework, which provides underlying support for communications, scalability, management of threads and sockets, efficient garbage collection, and server-related support.
Standards compliant	Compliance with the Java^{TM} 2 Platform Standard Edition 5.0 API Specification
Failover and Loadbalancing	WebLogic Server support for failover and loadbalancing of RMI objects.
WebLogic RMI compiler	Stubs and skeletons dynamically generated by WebLogic RMI at run time, which obviates need to explicitly run `weblogic.rmic`, except for clusterable or Internet Inter-ORB Protocol (IIOP) clients.
Dynamic Proxies	A class used by the clients of a remote object. In the case of RMI, skeleton and a stub classes are used. The stub class is the instance that is invoked upon in the client's Java Virtual Machine (JVM). The skeleton class, which exists in the remote JVM, unmarshals the invoked method and arguments on the remote JVM, invokes the method on the instance of the remote object, and then marshals the results for return to the client.
Security Support	No Security Manager required. WebLogic Server implements authentication, authorization, and Java EE security services.
Transaction Support	WebLogic Server supports transactions in the Sun Microsystems, Inc., Java Platform, Enterprise Edition (Java EE) programming model.
Internet Protocol version 6 (IPv6) Support	Support for 128 bit addressing space.

WebLogic RMI Features

The following sections describe the WebLogic RMI features and guidelines required to program RMI for use with WebLogic Server:

WebLogic RMI Overview

WebLogic RMI is divided between a client and server framework. The client run time does not have server sockets and therefore does not listen for connections. It obtains its connections through the server. Only the server knows about the client socket. Therefore if you plan to host a remote object on the client, you must connect the client to WebLogic Server. WebLogic Server processes requests for and passes information to the client. In other words, client-side RMI objects can only be reached through a single WebLogic Server, even in a cluster. If a client-side RMI object is bound into the JNDI naming service, it only be reachable as long as the server that carried out the bind is reachable.

WebLogic RMI Security Support

WebLogic Server implements authentication, authorization, and Java EE security services. For more information see Introduction to Programing WebLogic Security at *Programming WebLogic Security*.

WebLogic RMI Transaction Support

WebLogic Server supports transactions in the Sun Microsystems, Inc., Java Platform, Enterprise Edition (Java EE) programming model. For detailed information on using transactions in WebLogic RMI applications, see the following:

- Transactions in WebLogic Server RMI Applications in *Programing WebLogic JTA* provides an overview on how transactions are implemented in WebLogic RMI applications.

- Transactions in RMI Applications in *Programing WebLogic JTA* provides general guidelines when implementing transactions in RMI applications for WebLogic Server.

Failover and Load Balancing RMI Objects

The following sections contain information on WebLogic Server support for failover and loadbalancing of RMI objects:

- Clustered RMI Applications

- Load Balancing RMI Objects

- Parameter-Based Routing for Clustered Objects

Clustered RMI Applications

For clustered RMI applications, failover is accomplished using the object's replica-aware stub. When a client makes a call through a replica-aware stub to a service that fails, the stub detects the failure and retries the call on another replica.

To make Java EE services available to a client, WebLogic binds an RMI stub for a particular service into its JNDI tree under a particular name. The RMI stub is updated with the location of other instances of the RMI object as the instances are deployed to other servers in the cluster. If a server within the cluster fails, the RMI stubs in the other server's JNDI tree are updated to reflect the server failure.

You specify the generation of replica-aware stubs for a specific RMI object using the `-clusterable` option of the WebLogic RMI compiler. For example:

```
$ java weblogic.rmic -clusterable classes
```

For more information, see Replication and Failover for EJBs and RMIs in *Using WebLogic Cluster*s.

Load Balancing RMI Objects

The load balancing algorithm for an RMI object is maintained in the replica-aware stub obtained for a clustered object. You specify the load balancing algorithm for a specific RMI object using the `-loadAlgorithm <algorithm>` option of the WebLogic RMI compiler. A load balancing algorithm that you configure for an object overrides the default load balancing algorithm for the cluster. The WebLogic Server RMI compiler supports the following load balancing algorithms:

- Round Robin Load Balancing

- Weight-Based Load Balancing

- Random Load Balancing

- Server Affinity Load Balancing Algorithms

For example:

To set load balancing on an RMI object to round robin, use the following `rmic` options:

```
$ java weblogic.rmic -clusterable -loadAlgorithm round-robin classes
```

To set load balancing on an RMI object to weight-based server affinity, use `rmic` options:

```
$ java weblogic.rmic -clusterable -loadAlgorithm weight-based -stickToF
irstServer classes
```

For more information, see Load Balancing for EJBs and RMI Objects in *Using WebLogic Server Clusters*.

Parameter-Based Routing for Clustered Objects

Parameter-based routing allows you to control load balancing behavior at a lower level. Any clustered object can be assigned a CallRouter using the `weblogic.rmi.cluster.CallRouter` interface. This is a class that is called before each invocation with the parameters of the call. The

CallRouter is free to examine the parameters and return the name server to which the call should be routed.

```
weblogic.rmi.cluster.CallRouter.

Class java.lang.Object
   Interface weblogic.rmi.cluster.CallRouter
   (extends java.io.Serializable)
```

A class implementing this interface must be provided to the RMI compiler (`rmic`) to enable parameter-based routing. Run `rmic` on the service implementation using these options (to be entered on one line):

```
$ java weblogic.rmic -clusterable -callRouter <callRouterClass> <remote
ObjectClass>
```

The call router is called by the clusterable stub each time a remote method is invoked. The router is responsible for returning the name of the server to which the call should be routed.

Each server in the cluster is uniquely identified by its name as defined with the WebLogic Server Console. These are the names that the method router must use for identifying servers.

Example: Consider the `ExampleImpl` class which implements a remote interface Example, with one method `foo`:

```
public class ExampleImpl implements Example {
public void foo(String arg) { return arg; }
}
```

This `CallRouter` implementation `ExampleRouter` ensures that all `foo` calls with 'arg' < "n" go to server1 (or server3 if server1 is unreachable) and that all calls with 'arg' >= "n" go to server2 (or server3 if server2 is unreachable).

```
public class ExampleRouter implements CallRouter {
  private static final String[] aToM = { "server1", "server3" };
  private static final String[] nToZ = { "server2", "server3" };

  public String[] getServerList(Method m, Object[] params) {
    if (m.GetName().equals("foo")) {
      if (((String)params[0]).charAt(0) < 'n') {
        return aToM;
```

```
        } else {
          return nToZ;
        }
      } else {
        return null;
      }
    }
}
```

This `rmic` call associates the `ExampleRouter` with `ExampleImpl` to enable parameter-based routing:

```
$ rmic -clusterable -callRouter ExampleRouter ExampleImpl
```

Custom Call Routing and Collocation Optimization

If a replica is available on the same server instance as the object calling it, the call is not load-balanced as it is more efficient to use the local replica. For more information, see Optimization for Collocated Objects in *Using WebLogic Server Clusters*.

Creating Pinned Services

You can also use `weblogic.rmic` to generate stubs that are *not* replicated in the cluster. These stubs are known as "pinned" services, because after they are registered they are available only from the host with which they are registered and will not provide transparent failover or load balancing. Pinned services are available cluster-wide, because they are bound into the replicated cluster-wide JNDI tree. However, if the individual server that hosts the pinned services fails, the client cannot failover to another server.

You specify the generation of non-replicated stubs for a specific RMI object by **not** using the `-clusterable` option of the WebLogic RMI compiler. For example:

```
$ java weblogic.rmic classes
```

Dynamic Proxies in RMI

A *dynamic proxy* or *proxy* is a class used by the clients of a remote object. This class implements a list of interfaces specified at runtime when the class is created. In the case of RMI, *dynamically generated bytecode* and *proxy* classes are used. The proxy class is the instance that is invoked upon in the client's Java Virtual Machine (JVM). The proxy class marshals the invoked method

name and its arguments; forwards these to the remote JVM. After the remote invocation is completed and returns, the proxy class unmarshals the results on the client. The generated bytecode—which exists in the remote JVM—unmarhsals the invoked method and arguments on the remote JVM, invokes the method on the instance of the remote object, and then marshals the results for return to the client.

Using the RMI Timeout

WebLogic Server allows you to specify a timeout for synchronous remote call. This allows an RMI client making a remote call to return before the remote method that it invoked has returned from the server instance it called. This can be useful in legacy applications where a client wants to be able to return quickly if there is no response from the remote system. See "Guidelines on Using the RMI Timeout" on page 9-48.

To implement a synchronous RMI timeout, use the `remote-client-timeout` deployment descriptor element found in the `weblogic-ejb-jar.xml`. For more information, see the weblogic-ejb-jar.xml Deployment Descriptor Reference in *Programming WebLogic Enterprise JavaBeans*.

Using the WebLogic RMI Compiler

The following sections describe the WebLogic RMI compiler:

- Overview of the WebLogic RMI Compiler
- WebLogic RMI Compiler Features
- WebLogic RMI Compiler Options

Overview of the WebLogic RMI Compiler

The WebLogic RMI compiler (`weblogic.rmic`) is a command-line utility for generating and compiling remote objects. Use `weblogic.rmic` to generate dynamic proxies on the client-side for custom remote object interfaces in your application and provide hot code generation for server-side objects.

You only need to explicitly run `weblogic.rmic` for clusterable or IIOP clients. 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). See Chapter 7, "Using RMI over IIOP."

WebLogic RMI Compiler Features

The following sections provide information on WebLogic RMI Compiler features for this release:

- Hot Code Generation

- Proxy Generation

- Additional WebLogic RMI Compiler Features

Hot Code Generation

When you run `rmic`, you use WebLogic Server's hot code generation feature to automatically generate bytecode in memory for server classes. This bytecode is generated on the fly as needed for the remote object. WebLogic Server no longer generates the skeleton class for the object when `weblogic.rmic` is run.

Hot code generation produces the bytecode for a server-side class that processes requests from the dynamic proxy on the client. The dynamically created bytecode de-serializes client requests and executes them against the implementation classes, serializing results and sending them back to the proxy on the client. The implementation for the class is bound to a name in the WebLogic RMI registry in WebLogic Server.

Proxy Generation

The default behavior of the WebLogic RMI compiler is to produce proxies for the *remote interface* and for the remote classes to share the proxies. A *proxy* is a class used by the clients of a remote object. In the case of RMI, *dynamically generated bytecode* and *proxy* classes are used.

For example, `example.hello.HelloImpl` and `counter.example.CiaoImpl` are represented by a single proxy class and bytecode—the proxy that matches the remote interface implemented by the remote object, in this case, `example.hello.Hello`.

When a remote object implements more than one interface, the proxy names and packages are determined by encoding the set of interfaces. You can override this default behavior with the WebLogic RMI compiler option `-nomanglednames`, which causes the compiler to produce proxies specific to the remote class. When a class-specific proxy is found, it takes precedence over the interface-specific proxy.

In addition, with WebLogic RMI proxy classes, the proxies are not final. References to collocated remote objects are references to the objects themselves, not to the proxies.

The dynamic proxy class is the serializable class that is passed to the client. A client acquires the proxy for the class by looking up the class in the WebLogic RMI registry. The client calls methods on the proxy just as if it were a local class and the proxy serializes the requests and sends them to WebLogic Server.

Additional WebLogic RMI Compiler Features

Other features of the WebLogic RMI compiler include the following:

- Signatures of remote methods do not need to throw `RemoteException`.

- Remote exceptions can be mapped to `RuntimeException`.

- Remote classes can also implement non-remote interfaces.

WebLogic RMI Compiler Options

The WebLogic RMI compiler accepts any option supported by the Java compiler; for example, you could add `-d \classes examples.hello.HelloImpl` to the compiler option at the command line. All other options supported by the Java compiler can be used and are passed directly to the Java compiler.

The following table lists `java weblogic.rmic` options. Enter these options after `java weblogic.rmic` and before the name of the remote class.

```
$java weblogic.rmic [options] <classes>...
```

Table 4-1 WebLogic RMI Compiler Options

Option	Description
-help	Prints a description of the options.
-version	Prints version information.
-d <dir>	Specifies the target (top level) directory for compilation.
-dispatchPolicy <queueName>	Specifies a configured execute queue that the service should use to obtain execute threads in WebLogic Server.
-oneway	Specifies all calls are one-way calls.
-idl	Generates IDLs for remote interfaces.
-idlOverwrite	Overwrites existing IDL files.
-idlVerbose	Displays verbose information for IDL information.
-idlDirectory <idlDirectory>	Specifies the directory where IDL files will be created (Default = current directory).

Table 4-1 WebLogic RMI Compiler Options

Option	Description
`-idlFactories`	Generates factory methods for valuetypes.
`-idlNoValueTypes`	Prevents the generation of valuetypes and the methods/attributes that contain them.
`-idlNoAbstractInterfaces`	Prevents the generation of abstract interfaces and the methods/attributes that contain them.
`-idlStrict`	Generates IDL according to OMG standard.
`-idlVisibroker`	Generate IDL compatible with Visibroker 4.5 C++.
`-idlOrbix`	Generate IDL compatible with Orbix 2000 2.0 C++.
`-iiopTie`	Generate CORBA skeletons using Sun's version of `rmic`.
`-iiopSun`	Generate CORBA stubs using Sun's version of `rmic`.
`-nontransactional`	Suspends the transaction before making the RMI call and resumes after the call completes.
`-compiler <javac>`	Specifies the Java compiler. If not specified, the `-compilerclass` option will be used.
`-compilerclass <com.sun.tools.javac.Main>`	Compiler class to invoke.
`-clusterable`	This cluster-specific options marks the service as clusterable (can be hosted by multiple servers in a WebLogic Server cluster). Each hosting object, or replica, is bound into the naming service under a common name. When the service stub is retrieved from the naming service, it contains a replica-aware reference that maintains the list of replicas and performs load-balancing and fail-over between them.
`-loadAlgorithm <algorithm>`	Only for use in conjunction with `-clusterable`. Specifies a service-specific algorithm to use for load-balancing and fail-over (Default = `weblogic.cluster.loadAlgorithm`). Must be one of the following: round-robin, random, or weight-based.

Table 4-1 WebLogic RMI Compiler Options

Option	Description
`-callRouter` `<callRouterClass>`	This cluster-specific option used in conjunction with `-clusterable` specifies the class to be used for routing method calls. This class must implement `weblogic.rmi.cluster.CallRouter`. If specified, an instance of the class is called before each method call and can designate a server to route to based on the method parameters. This option either returns a server name or null. Null means that you use the current load algorithm.
`-stickToFirstServer`	This cluster-specific option used in conjunction with `-clusterable` enables "sticky" load balancing. The server chosen for servicing the first request is used for all subsequent requests.
`-methodsAreIdempotent`	This cluster-specific option used in conjunction with `-clusterable` indicates that the methods on this class are idempotent. This allows the stub to attempt recovery from any communication failure, even if it can not ensure that failure occurred before the remote method was invoked. By default (if this option is not used), the stub only retries on failures that are guaranteed to have occurred before the remote method was invoked.
`-iiop`	Generates IIOP stubs from servers.
`-iiopDirectory`	Specifies the directory where IIOP proxy classes are written.
`-timeout`	Used in conjunction with remote-client-timeout.
`-commentary`	Emits commentary.
`-nomanglednames`	Causes the compiler to produce proxies specific to the remote class.
`-g`	Compile debugging information into the class.
`-O`	Compile with optimization.
`-nowarn`	Compile without warnings.
`-verbose`	Compile with verbose output.

Table 4-1 WebLogic RMI Compiler Options

Option	Description
`-verboseJavac`	Enable Java compiler verbose output.
`-nowrite`	Prevent the generation of `.class` files.
`-deprecation`	Provides warnings for deprecated calls.
`-classpath <path>`	Specifies the classpath to use.
`-J<option>`	Use to pass flags through to the Java runtime.
`-keepgenerated`	Allows you to keep the source of generated stub and skeleton class files when you run the WebLogic RMI compiler.
`-disableHotCodeGen`	Causes the compiler to create stubs at skeleton classes when compiled.

Non-Replicated Stub Generation

You can also use `weblogic.rmic` to generate stubs that are *not* replicated in the cluster. These stubs are known as "pinned" services, because after they are registered they are available only from the host with which they are registered and will not provide transparent failover or load balancing. Pinned services are available cluster-wide, because they are bound into the replicated cluster-wide JNDI tree. However, if the individual server that hosts the pinned services fails, the client cannot failover to another server.

Using Persistent Compiler Options

During deployment, appc and ejbc run each EJB container class through the RMI compiler to create RMI descriptors necessary to dynamically generate stubs and skeletons. Use the `weblogic-ejb-jar.xml` file to persist `iiop-security-descriptor` elements. For more information, see 2.1 weblogic-ejb-jar.xml Elements in *Programming WebLogic Enterprise JavaBeans.*

Using WebLogic RMI with T3 Protocol

The following sections provide information on using WebLogic RMI with T3 protocol.

- "RMI Communication in WebLogic Server" on page 5-21
- "Determining Connection Availability" on page 5-21

RMI Communication in WebLogic Server

RMI communications in WebLogic Server use the T3 protocol to transport data between WebLogic Server and other Java programs, including clients and other WebLogic Server instances. A server instance keeps track of each Java Virtual Machine (JVM) with which it connects, and creates a single T3 connection to carry all traffic for a JVM. See "Configure T3 protocol" in *Administration Console Online Help*.

For example, if a Java client accesses an enterprise bean and a JDBC connection pool on WebLogic Server, a single network connection is established between the WebLogic Server JVM and the client JVM. The EJB and JDBC services can be written as if they had sole use of a dedicated network connection because the T3 protocol invisibly multiplexes packets on the single connection.

Determining Connection Availability

Any two Java programs with a valid T3 connection—such as two server instances, or a server instance and a Java client—use periodic point-to-point "heartbeats" to announce and determine

continued availability. Each end point periodically issues a heartbeat to the peer, and similarly, determines that the peer is still available based on continued receipt of heartbeats from the peer.

- The frequency with which a server instance issues heartbeats is determined by the *heartbeat interval*, which by default is 60 seconds.

- The number of missed heartbeats from a peer that a server instance waits before deciding the peer is unavailable is determined by the *heartbeat period*, which by default, is 4. Hence, each server instance waits up to 240 seconds, or 4 minutes, with no messages— either heartbeats or other communication—from a peer before deciding that the peer is unreachable.

- Changing timeout defaults is not recommended.

How to Implement WebLogic RMI

The basic building block for all remote objects is the interface `java.rmi.Remote`, which contains no methods. You extend this "tagging" interface—that is, it functions as a tag to identify remote classes—to create your own remote interface, with method stubs that create a structure for your remote object. Then you implement your own remote interface with a remote class. This implementation is bound to a name in the registry, where a client or server can look up the object and use it remotely.

If you have written RMI classes, you can drop them in WebLogic RMI by changing the import statement on a remote interface and the classes that extend it. To add remote invocation to your client applications, look up the object by name in the registry. WebLogic RMI exceptions are identical to and extend `java.rmi` exceptions so that existing interfaces and implementations do not have to change exception handling.

Procedures for Implementing WebLogic RMI

The following sections describe how to implement WebLogic Server RMI:

- Creating Classes That Can Be Invoked Remotely

    Step 1. Write a Remote Interface

    Step 2. Implement the Remote Interface

    Step 3. Compile the Java Class

    Step 4. Compile the Implementation Class with RMI Compiler

    Step 5: Write Code That Invokes Remote Methods

- Hello Code Sample

Creating Classes That Can Be Invoked Remotely

You can write your own WebLogic RMI classes in just a few steps. Here is a simple example.

Step 1. Write a Remote Interface

Every class that can be remotely invoked implements a remote interface. Using a Java code text editor, write the remote interface in adherence with the following guidelines.

- A remote interface must extend the interface `java.rmi.Remote`, which contains no method signatures. Include method signatures that will be implemented in every remote class that implements the interface. For detailed information on how to write an interface, see the Sun Microsystems JavaSoft tutorial Creating Interfaces.

- The remote interface must be public. Otherwise a client gets an error when attempting to load a remote object that implements it.

- Unlike the JavaSoft RMI, it is not necessary for each method in the interface to declare `java.rmi.RemoteException` in its `throws` block. The exceptions that your application throws can be specific to your application, and can extend `RuntimeException`. WebLogic RMI subclasses `java.rmi.RemoteException,` so if you already have existing RMI classes, you will not have to change your exception handling.

- Your Remote interface may not contain much code. All you need are the method signatures for methods you want to implement in remote classes.

  Here is an example of a remote interface with the method signature `sayHello()`.

```
package examples.rmi.multihello;

import java.rmi.*;

public interface Hello extends java.rmi.Remote {

  String sayHello() throws RemoteException;

}
```

With JavaSoft's RMI, every class that implements a remote interface must have accompanying, precompiled proxies. WebLogic RMI supports more flexible runtime code generation; WebLogic RMI supports dynamic proxies and dynamically created bytecode that are type-correct but are otherwise independent of the class that implements the interface. If a class implements a single remote interface, the proxy and bytecode that is generated by the compiler will have the same name as the remote interface. If a class implements more than one remote interface, the name of

the proxy and bytecode that result from the compilation depend on the name mangling used by the compiler.

Step 2. Implement the Remote Interface

Still using a Java code text editor, write the class be invoked remotely. The class should implement the remote interface that you wrote in Step 1, which means that you implement the method signatures that are contained in the interface. Currently, all the code generation that takes place in WebLogic RMI is dependent on this class file.

With WebLogic RMI, your class does not need to extend `UnicastRemoteObject`, which is required by JavaSoft RMI. (You can extend `UnicastRemoteObject`, but it isn't necessary.) This allows you to retain a class hierarchy that makes sense for your application.

Note: With Weblogic server, you can use both Weblogic RMI and standard JDK RMI. If you use Weblogic RMI, then you must use "`java weblogic.rmic ...`" as the rmic compiler and you must not create your RMI implementation as a subclass of "`java.rmi.server.UnicastRemoteObject`". If you use standard JDK RMI, then you must use "`%JAVA_HOME%\bin\rmic`" as the rmic compiler and you must create your RMI implementation class as a subclass of "`java.rmi.server.UnicastRemoteObject`".

Your class can implement more than one remote interface. Your class can also define methods that are not in the remote interface, but you cannot invoke those methods remotely.

This example implements a class that creates multiple `HelloImpls` and binds each to a unique name in the registry. The method `sayHello()` greets the user and identifies the object which was remotely invoked.

```
package examples.rmi.multihello;

import java.rmi.*;

public class HelloImpl implements Hello {

  private String name;

  public HelloImpl(String s) throws RemoteException {

    name = s;

  }

  public String sayHello() throws RemoteException {

    return "Hello! From " + name;

  }
```

Next, write a main() method that creates an instance of the remote object and registers it in the WebLogic RMI registry, by binding it to a name (a URL that points to the implementation of the object). A client that needs to obtain a proxy to use the object remotely will be able to look up the object by name.

Below is an example of a main() method for the HelloImpl class. This registers the HelloImpl object under the name HelloRemoteWorld in a WebLogic Server registry.

```
public static void main(String[] argv) {

  // Not needed with WebLogic RMI

  // System.setSecurityManager(new RmiSecurityManager());

  // But if you include this line of code, you should make

  // it conditional, as shown here:

  // if (System.getSecurityManager() == null)

  //   System.setSecurityManager(new RmiSecurityManager());

  int i = 0;

  try {

    for (i = 0; i < 10; i++) {

      HelloImpl obj = new HelloImpl("MultiHelloServer" + i);

      Context.rebind("//localhost/MultiHelloServer" + i, obj);

     System.out.println("MultiHelloServer" + i + " created.");

    }

    System.out.println("Created and registered " + i +

                      " MultiHelloImpls.");

  }

  catch (Exception e) {

    System.out.println("HelloImpl error: " + e.getMessage());

    e.printStackTrace();

  }

}
```

WebLogic RMI does not require that you set a Security Manager in order to integrate security into your application. Security is handled by WebLogic Server support for SSL and ACLs. If you must, you may use your own security manager, but do not install it in WebLogic Server.

Step 3. Compile the Java Class

Use `javac` or some other Java compiler to compile the `.java` files to produce `.class` files for the remote interface and the class that implements it.

Step 4. Compile the Implementation Class with RMI Compiler

Run the WebLogic RMI compiler (`weblogic.rmic`) against the remote class to generate the dynamic proxy and bytecode, on the fly. A proxy is the client-side proxy for a remote object that forwards each WebLogic RMI call to its matching server-side bytecode, which in turn forwards the call to the actual remote object implementation. To run the `weblogic.rmic`, use the command pattern:

```
$ java weblogic.rmic nameOfRemoteClass
```

where `nameOfRemoteClass` is the full package name of the class that implements your Remote interface. With the examples we have used previously, the command would be:

```
$ java weblogic.rmic examples.rmi.hello.HelloImpl
```

Set the flag `-keepgenerated` when you run `weblogic.rmic` if you want to keep the generated source when creating stub or skeleton classes. For a listing of the available command-line options, see "WebLogic RMI Compiler Options" on page 4-17.

Step 5: Write Code That Invokes Remote Methods

Using a Java code text editor, once you compile and install the remote class, the interface it implements, and its proxy and the bytecode on the WebLogic Server, you can add code to a WebLogic client application to invoke methods in the remote class.

In general, it takes just a single line of code: get a reference to the remote object. Do this with the `Naming.lookup()` method. Here is a short WebLogic client application that uses an object created in a previous example.

```
package mypackage.myclient;

import java.rmi.*;


public class HelloWorld throws Exception {
```

```
// Look up the remote object in the

// WebLogic's registry

Hello hi = (Hello)Naming.lookup("HelloRemoteWorld");

// Invoke a method remotely

String message = hi.sayHello();

System.out.println(message);

}
```

This example demonstrates using a Java application as the client.

Hello Code Sample

Here is the full code for the Hello interface.

```
package examples.rmi.hello;

import java.rmi.*;


public interface Hello extends java.rmi.Remote {


  String sayHello() throws RemoteException;


}
```

Here is the full code for the `HelloImpl` class that implements it.

```
package examples.rmi.hello;


import java.rmi.*;


public class HelloImpl

    // Don't need this in WebLogic RMI:

    // extends UnicastRemoteObject
```

```
    implements Hello {

  public HelloImpl() throws RemoteException {
    super();
  }

  public String sayHello() throws RemoteException {
    return "Hello Remote World!!";
  }

  public static void main(String[] argv) {
    try {
      HelloImpl obj = new HelloImpl();
      Naming.bind("HelloRemoteWorld", obj);
    }
    catch (Exception e) {
      System.out.println("HelloImpl error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Using RMI over IIOP

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

- What is RMI over IIOP?

- Overview of WebLogic RMI-IIOP

- Protocol Compatibility

What is RMI over IIOP?

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

Overview of WebLogic RMI-IIOP

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

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

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

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

- Interoperate between WebLogic Server and Tuxedo clients

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

How you develop your RMI-IIOP applications depends on what services and clients you are trying to integrate. See Programming Stand-alone Clients for more information on how to create applications for various clients types that use RMI and RMI-IIOP.

Figure 7-1 shows RMI Object Relationships for objects that use IIOP.

Figure 7-1   RMI Object Relationships
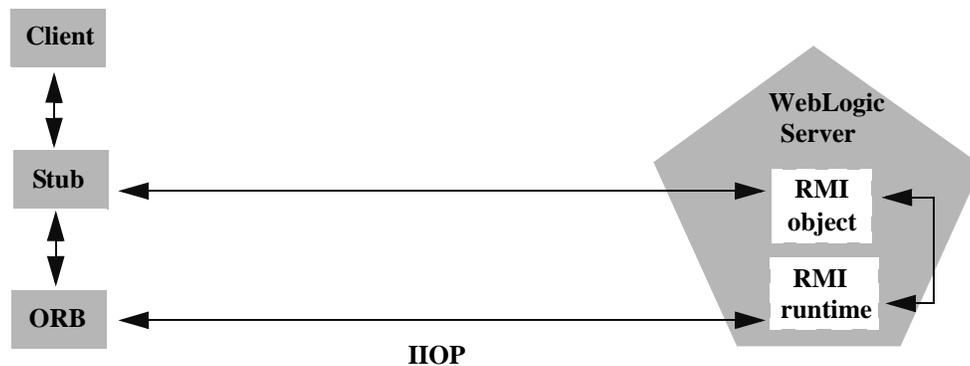
Support for RMI-IIOP with RMI (Java) Clients

You can use RMI-IIOP with Java/RMI clients, taking advantage of the standard IIOP protocol. WebLogic Server provides multiple options for using RMI-IIOP in a Java-to-Java environment, including the new Java EE Application Client (thin client), which is based on the new small footprint client jar. To use the new thin client, you need to have the `wlclient.jar` (located in `WL_HOME/server/lib`) on the client side's CLASSPATH. For more information on RMI-IIOP client options, see Programming Stand Alone Clients.

Support for RMI-IIOP with Tuxedo Client

WebLogic Server contains an implementation of the WebLogic Tuxedo Connector, an underlying technology that enables you to interoperate with Tuxedo servers. Using WebLogic Tuxedo Connector, you can leverage Tuxedo as an ORB, or integrate legacy Tuxedo systems with applications you have developed on WebLogic Server. For more information, see the *WebLogic Tuxedo Connector Programmer's Guide*.

Support for RMI-IIOP with CORBA/IDL Clients

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

Protocol Compatibility

Interoperability between this release and prior WebLogic Server releases 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 7-1 WebLogic Server-to-Server Interoperability

To Server	WebLogic Server 8.1	WebLogic Server 9.x	WebLogic Server 10.0
From Server			
WebLogic Server 8.1	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services
WebLogic Server 9.x	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services
WebLogic Server 10.0	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services
Sun JDK ORB client[1]	RMI/IIOP[2]	RMI/IIOP[3]	RMI/IIOP[4]

1. This option involves calling directly into the JDK ORB from within application hosted on WebLogic Server.
2. JDK 1.3.x or 1.4.1. No clustering. No transaction propagation
3. JDK 5.0. No clustering. No transaction propagation
4. JDK 5.0. 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 7-2 Client-to-Server Interoperability

To Server	WebLogic Server 8.1	WebLogic Server 9.x	WebLogic Server 10.0
From Client (stand-alone)			
WebLogic Server 8.1	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services
WebLogic Server 9.x	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services
WebLogic Server 10.0	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services	RMI/T3 RMI/IIOP HTTP Web Services
Sun JDK ORB client[1]	RMI/IIOP[2]	RMI/IIOP[3]	RMI/IIOP[4]

1. This option involved calling directly into the JDK ORB from within a client application.
2. JDK 1.3.x or 1.4.1. No clustering. No transaction propagation
3. JDK 5.0. No clustering. No transaction propagation
4. JDK 5.0. No clustering. No transaction propagation

Configuring WebLogic Server for RMI-IIOP

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

- Set the Listening Address

- Setting Network Channel Addresses

- Using a IIOPS Thin Client Proxy

- Using RMI-IIOP with SSL and a Java Client

- Accessing WebLogic Server Objects from a CORBA Client through Delegation

- "Configuring CSIv2 authentication" on page 8-43

- Using RMI over IIOP with a Hardware LoadBalancer

- Limitations of WebLogic RMI-IIOP

- Propagating Client Identity

Set the Listening Address

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

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

as described in "Configuring Network Resources" in *Configuring WebLogic Server Environments*.

Setting Network Channel Addresses

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

Considerations for Proxys and Firewalls

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

Considerations for Clients with Multiple Connections

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

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

      -Dweblogic.corba.client.bidir=true

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

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

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

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

Using a IIOPS Thin Client Proxy

The IIOPs Thin Client Proxy provides a WebLogic thin client the ability to proxy outbound requests to a server. In this situation, each user routes all outbound requests through their proxy. The user's proxy then directs the request to the WebLogic Server. You should use this method

when it is not practical to implement a Network Channel. To enable a proxy, set the following properties:

```
-Diiops.proxyHost=<host>

-Diiops.proxyPort=<port>
```

where:

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

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

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

```
-Diiops.proxyHost=https.proxyHost

-Diiops.proxyPort=https.proxyPort
```

You should consider the following security implications:

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

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

Using RMI-IIOP with SSL and a Java Client

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

Accessing WebLogic Server Objects from a CORBA Client through Delegation

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

Overview of Delegation

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

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

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

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

Figure 8-1 is a diagram of a CORBA client invoking an EJB by delegating the call to an implementation class that connects to the server and operates upon the EJB. Using a similar architecture, the reverse situation will also work. You can have a startup class that brings up an ORB and obtains a reference to the CORBA implementation object of interest. This class can make itself available to other WebLogic objects throughout the JNDI tree and delegate the appropriate calls to the CORBA object.

Figure 8-1   CORBA Client Invoking an EJB with a Delegated Call

Delegate architecture
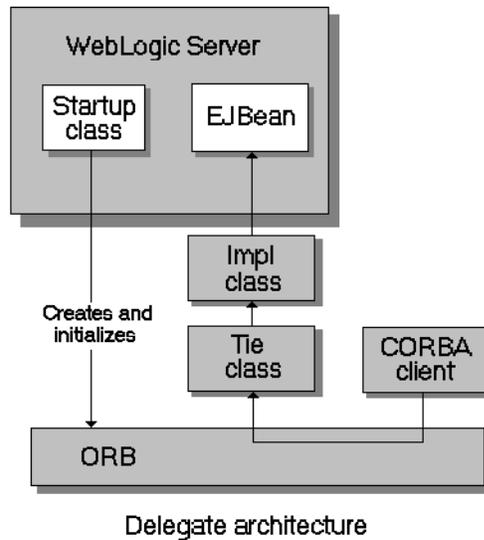
Example of Delegation

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

- Creating the ORB

- Creating the Tie object

- Associating the implementation class with the Tie object

- Registering the Tie object with the ORB

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

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

```java
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();

    }

  }

}
```

Configuring CSIv2 authentication

The Common Secure Interoperability Specification, Version 2 (CSIv2) is an Open Management Group (OMG) specification that addresses the requirements of Common Object Request Broker Architecture (CORBA) security for interoperable authentication, delegation, and privileges. See Common Secure Interoperability Version 2 (CSIv2) in *Understanding WebLogic Security*.

Use the following steps to use CSIv2 to authenticate an inbound call from a remote domain:

1. Update the Identity Asserter. See "Configuring Identity Assertion Providers" in *Securing WebLogic Server.*

2. Update the User Name Mapper. See "Configuring a User Name Mapper" in in *Securing WebLogic Server.*

3. Add all users required by the application in the remote domain to the WebLogic AuthenticationProvider. See "Create User" in *Administration Console Online Help*.

Using RMI over IIOP with a Hardware LoadBalancer

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

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

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

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

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

- false—Default value.

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

Limitations of WebLogic RMI-IIOP

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

Limitations Using RMI-IIOP on the Client

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

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

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

- Have bugs in its treatment of mangled method names.

- Do not correctly unmarshal unchecked exceptions.

- Have subtle bugs relating to the encoding of valuetypes.

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

Limitations Developing Java IDL Clients

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

Limitations of Passing Objects by Value

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

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

Simply running `ejbc` on the defined EJB interfaces will **not** generate these definitions because they do not appear in the interface. For this reason `ejbc` will also accept Java classes that are not remote interfaces—specifically for the purpose of generating IDL for these interfaces. Review the `/iiop/ejb/entity/cppclient` example to see how to register a value type factory.

Java types that are serializable but that define `writeObject()` are mapped to custom value types in IDL. You must write C++ code to unmarshal the value type manually. See example code from the `iiop/ejb/entity/tuxclient/ArrayList_i.cpp` file at http://dev2dev.bea.com/.

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

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

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

Propagating Client Identity

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

- The identity of any client connecting over IIOP to WebLogic Server will default to `<anonymous>`. You can set the user and password in the `config.xml` file to establish a

single identity for all clients connecting over IIOP to a particular instance of WebLogic Server, as shown in the example below:

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

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

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

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

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

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

These methods are not required when using `WLInitialContextFactory` in RMI clients or can be avoided by using the WebLogic C++ client. See example code from the `iiop/ejb/stateless/sectuxclient` example at http://dev2dev.bea.com/.

Best Practices for Application Design

The following sections discuss recommended design patterns when programming with RMI and RMI over IIOP:

Use java.rmi

BEA recommends RMI users use `java.rmi`. Although the WebLogic API contains the `weblogic.rmi` API, it is deprecated and is only provided as a compatibility API. Other WebLogic APIs provided for compatibility are :

- `weblogic.rmi.registry`

- `weblogic.rmi.server`

- `weblogic.rmi.extensions`

Use PortableRemoteObject

To maintain code portability, always use `PortableRemoteObject` when casting the home interfaces. For example:

```
Propshome home = (PropsHome)
```

```
PortableRemoteObject.narrow(

    ctx.lookup( "Props" ),

    PropsHome.class );
```

Use WebLogic Work Areas

Work Contexts allow Java EE developers to define properties as application context which implicitly flow across remote requests and allow downstream components to work in the context of the invoking client. Work Contexts allow developers to pass properties without including them in a remote call. A Work Context is propagated with each remote call-allowing the called component to add or modify properties defined in the Work Context; similarly, the calling component can access the Work Context to obtain new or updated properties.

Work Contexts ease the processing of implementing and maintaining functionality that requires that information to be passed to remote components, such as diagnostics monitoring, application transactions, and application load-balancing. Work Contexts are also a useful mechanism for providing information to third-party components.

Work Conexts can propagate user-defined properties across all request scopes supported by WebLogic Server-a Work Context is available to all of the objects that can exist within the request scope, including RMI calls. For more information, see Developing Applications with WebLogic Server.

Guidelines on Using the RMI Timeout

This feature provides a work around for legacy systems where the behavior of asynchronous calls is desired but not yet implemented. BEA recommends legacy systems implement more appropriate technologies if possible, such as:

- Asynchronous RMI invokations

- JMS and Message Driven Beans (MDBs)

- HTTP servlett applications

If you need to use the RMI timeout for a legacy sytem, review the following guidelines:

- The RMI timeout should be used only when the following three conditions are met:

    – The method call is idempotent or does not introduce any state change

    – The method call is non-transactional

– No JMS resources are involved in the call

- There is no transparent failover to another cluster node when a request times out. `RequestTimeOutException` is always propogated to the caller.

- The server continues to process requests that have timed out. The client is required check the state of the request on the server before reattempting the call.

- If a server times out, the client has the ability to mark the server as unreachable in the client side cluster reference. This prevents calls from being directed to the marked server for a specified time.

Best Practices for Application Design

CORBA Support for WebLogic Server

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

- "Specification References" on page A-1

- "Supported Specification Details" on page A-2

- "Tools" on page A-2

Specification References

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

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

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

- IDL to Java language mapping: ptc/03-09-04 at
  http://www.omg.org/cgi-bin/doc?ptc/03-09-04

- Revised IDL to Java language mapping 1.3: formal/00-11-03 at
  http://www.omg.org/cgi-bin/doc?formal/00-11-03

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

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

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

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

Supported Specification Details

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

- CORBA 2.6, chapters 1-3, 6-7, 13 and 15.

- Revised IDL to Java language mapping, section 1.21.8.2, the `orb.properties` file.

- CORBA 2.6, chapter 4 and 5, excepting details relevant to excluded features from other chapters, such as PortableInterceptors.

- CORBA 2.6, sections 10.6.1 and 10.6.2 are supported for repository IDs.

- CORBA 2.6, section 10.7 for `TypeCode` APIs.

- CORBA 2.6, chapter 11, Portable Object Adapter (POA) excepting details relevant to excluded features from other chapters, such as PortableInterceptors.

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

- The Interoperable Naming Service.

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

- Transaction Service 1.2.1, as defined by the EJB 2.1 specification.

Tools

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

- The IDL to Java compiler (`idlj`) is the one that comes bundled with J2SE 5.0 and is compliant with following specification references:

- CORBA 2.3.1, chapter 3 (IDL definition).

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

- CORBA 2.3.1, section 10.6.5 (pragmas).

- The IDL to Java mapping specification.

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

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

  - CORBA 2.6, chapters 5 and 6 (value types).

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

  - IDL generated by the `-idl` flag complies with CORBA 2.6 chapter 3.

CORBA Support for WebLogic Server