BEA WebLogic Server™

Programming WebLogic RMI

Version 8.1
Revised: June 28, 2006

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

4. Using the WebLogic RMI Compiler

5. RMI Communication within WebLogic Server

6. How to Implement WebLogic RMI

CHAPTER 1

Introduction and Roadmap

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

Document Scope and Audience

This document is written for application developers who want to build e-commerce applications using Remote Method Invocation (RMI) 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™ 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 and Guidelines," 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 5, "RMI Communication within WebLogic Server," provides information RMI communications between WebLogic Server and other Java programs, including clients and other WebLogic Servers. .
- Chapter 6, "How to Implement WebLogic RMI," provides a simple step by step example of how to implement WebLogic RMI.

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 WebLogic Server Applications* is a guide to developing WebLogic Server applications.
- *Programming WebLogic JNDI* is a guide using the WebLogic Java Naming and Directory Interface.
- *Programming WebLogic RMI over IIOP* is a guide to developing applications using RMI over Internet Interop-Orb-Protocol (IIOP).

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 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 8.1 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 8.1 Start menu.

Additional Examples Available for Download

Additional API examples for download at http://dev2dev.bea.com/code/certwls81.jsp. 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

The following sections provides information on new and changed features for release of WebLogic RMI:

- Compliance with the Java™ 2 Platform Standard Edition 1.4 API Specification

- WebLogic Server 8.1 releases SP4 and higher allow you to specify a timeout for a 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. For more information, see "Using the RMI Timeout" on page 3-12.

For more release-specific information on new and changed features, see these sections in *WebLogic Server 8.1 Release Notes*:

- "WebLogic Server 8.1 Features and Changes" lists new, changed, and deprecate features.
- "WebLogic Server 8.1 Known Issues" lists known problems by service pack, for all WebLogic Server APIs.
- "WebLogic Server Resolved Problems" lists problems solved by service pack, for all for all WebLogic Server APIs, including JMS.

CHAPTER 2

Understanding WebLogic RMI

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

- What is WebLogic RMI?
- 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 tables highlight 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™ 2 Platform Standard Edition 1.4 API Specification
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.
Failover and Loadbalancing	WebLogic Server support for failover and loadbalancing of RMI objects.
Security Support	No Security Manager required. WebLogic Server implements authentication, authorization, and J2EE security services.
Transaction Support	WebLogic Server supports transactions in the Sun Microsystems, Inc., Java™ 2, Enterprise Edition (J2EE) programming model.
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.

CHAPTER 3

WebLogic RMI Features and Guidelines

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

- WebLogic RMI Overview
- WebLogic RMI Security Support
- WebLogic RMI Transaction Support
- Failover and Load Balancing RMI Objects
- Creating Pinned Services
- Dynamic Proxies in RMI
- Using the RMI Timeout

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 J2EE 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™ 2, Enterprise Edition (J2EE) 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 J2EE 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> <remoteObj
ectClass>
```

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 will not be load-balanced, because 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 **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.

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*.

Guidelines on Using the RMI Timeout

This section provides implementation guidelines for appropriate use of the RMI client 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
- 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
- The client is required to handle the `RequestTimeOutException` and the management of state.
- 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.

- There is no transparent failover to another cluster node when a request times out. `RequestTimeOutException` is always propogated to the caller.
- Users may see server warning messages similar to:

```
<Warning> <RJVM> <BEA-000510> <Unsolicited response: 4,690>
```

CHAPTER 4

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
- Migrating from Stubs and Skeletons to Dynamic Proxies and Bytecode

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 Programming WebLogic RMI over IIOP for more information on using RMI over IIOP.

WebLogic RMI Compiler Features

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

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. See Using Execute Queues to Control Thread Usage for more information.
`-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.

Table 4-1 WebLogic RMI Compiler Options

Option	Description
`-idlDirectory <idlDirectory>`	Specifies the directory where IDL files will be created (Default = current directory).
`-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.

Table 4-1 WebLogic RMI Compiler Options

Option	Description
`-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.
`-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.
`-timeout`	Used in conjunction with remote-client-timeout.
`-iiop`	Generates IIOP stubs from servers.
`-iiopDirectory`	Specifies the directory where IIOP proxy classes are written.
`-commentary`	Emits commentary.
`-nomanglednames`	Causes the compiler to produce proxies specific to the remote class.

Table 4-1 WebLogic RMI Compiler Options

Option	Description
`-g`	Compile debugging information into the class.
`-O`	Compile with optimization.
`-nowarn`	Compile without warnings.
`-verbose`	Compile with verbose output.
`-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 2.0 weblogic-ejb-jar.xml Elements in *Programming WebLogic Enterprise JavaBeans*.

Migrating from Stubs and Skeletons to Dynamic Proxies and Bytecode

In previous versions of WebLogic Server, pre 6.1, running `weblogic.rmic` generated stubs on the client and skeleton code on the server-side. Now, dynamic proxies have replace generated stubs on the client-side and bytecode has replaced skeletons on the server-side. So, you no longer need to generate classes.

To enable pre-6.1 WebLogic RMI objects to run under later versions of WebLogic Server, rerun `weblogic.rmic` on those objects. This will generate the necessary proxies and bytecode that enable the deployed RMI object. See "Proxy Generation" on page 4-16, for more information on dynamic proxies.

If your remote objects are EJBs, rerun `weblogic.ejbc` again to enable pre-WebLogic Server 6.1 objects to work in the post-6.1 version. See "ejbc" in *Programming WebLogic Enterprise JavaBeans* for instructions on using `weblogic.ejbc`.

Rerunning either `weblogic.rmic` with one or more of the following parameters,`-oneway, -clusterable, -stickToFirstServer` or `weblogic.ejbc` on the remote object produces a deployment descriptor file for that object.

CHAPTER 5

RMI Communication within WebLogic Server

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

- Overview of RMI Communication in WebLogic Server
- Determining Connection Availability

Overview of RMI Communication in WebLogic Server

RMI communications in WebLogic Server use the T3 protocol, an optimized protocol used to transport data between WebLogic Server and other Java programs, including clients and other WebLogic Servers. 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.

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.

Note: It is recommended you do not change the timeout value as it is configured internally.

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:

- 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();
    }
  }
}
```