BEA WebLogic Server

Programming WebLogic RMI

BEA WebLogic Server 6.1
Document Date: June 24, 2002

Programming WebLogic RMI

Part Number	Date	Software Version
N/A	June 24, 2001	BEA WebLogic Server Version 6.1

Contents

About 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.

The document is organized as follows:

- Chapter 1, “Introducing WebLogic RMI,” is an overview of WebLogic RMI features and its architecture.
- Chapter 2, “Programming Considerations,” describes the features that you use to program RMI for WebLogic Server.
- Chapter 3, “Implementing WebLogic RMI,” describes the packages shipped as part of WebLogic RMI and provides procedures for implementing WebLogic RMI. (The public API includes the WebLogic implementation of the RMI base classes, the registry, and the server packages.)

Audience

This document is written for application developers who want to build e-commerce applications using the Remote Method Invocation (RMI) features. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

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

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server. In addition to this document you may want to review the Programming RMI over IIOP document. 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).

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String` *`CustomerName`*`;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

`[ ]`	Optional items in a syntax line. *Example*:
	`java utils.MulticastTest -n` *`name`* `-a` *`address`*
	`[-p` *`portnumber`*`] [-t` *`timeout`*`] [-s` *`send`*`]`

`|`	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

1 Introducing 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.

WebLogic implements the JavaSoft RMI specification. WebLogic RMI provides standards-based distributed object computing. WebLogic Server enables fast, reliable, large-scale network computing, and WebLogic RMI allows products, services, and resources to exist anywhere on the network but appear to the programmer and the end user as part of the local environment.

WebLogic RMI scales linearly under load, and execution requests can be partitioned into a configured number of server threads. Multiple server threads allow WebLogic Server to take advantage of latency time and available processors.

WebLogic RMI is completely standards-compliant. If you use another implementation of RMI, you can convert your programs by changing nothing more than the import statement. Differences exist between the JavaSoft reference implementation of RMI and the WebLogic RMI product; however, these differences are completely transparent to the developer.

In addition, WebLogic RMI is fully integrated with WebLogic Java Naming and Directory Interface (JNDI). Applications can be partitioned into meaningful name spaces by using either the JNDI API or the Registry interfaces in WebLogic RMI.

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

Like the JavaSoft reference implementation of RMI, WebLogic RMI provides transparent remote invocation in different JVMs. Remote interfaces and implementations that are written to the RMI specification can be used with WebLogic RMI without changes.

The following tables highlight important features of WebLogic implementation of RMI.

Table 1-1 WebLogic RMI Performance

Feature	WebLogic RMI
Overall performance	Enhanced by WebLogic RMI integration into the WebLogic Server framework, which provides underlying support for communications, management of threads and sockets, efficient garbage collection, and server-related support.
Scalability	Scales linearly under load. Scales dramatically better than JavaSoft RMI. Even relatively small, single-processor, PC-class servers can support more than 1,000 simultaneous RMI clients, depending on server workload and complexity of method calls.

Table 1-1 WebLogic RMI Performance

Feature	WebLogic RMI
Management of threads and sockets	Uses a single, asynchronous, bidirectional connection for WebLogic RMI client-to-network traffic. Same connection can support WebLogic JDBC requests or other services.
Serialization	Uses high-performance serialization, which offers a significant performance gain, even for one-time use of remote class.
Resolution of co-located objects	No performance penalty for co-located objects that are defined as remote. References to co-located "remote" objects resolved as direct references to the actual implementation object.
Processes for supporting services	WebLogic RMI registry replaces the RMI registry process. WebLogic RMI runs inside WebLogic Server. No additional processes needed.

Table 1-2 WebLogic RMI Ease of Use

Feature	WebLogic RMI
rmic	Proxies and bytecode dynamically generated by WebLogic RMI at run time, which obviates need to explicitly run rmic, except for clusterable or IIOP clients.
Ease-of-use extensions	Provides ease-of-use extensions for remote interfaces and code generation. For example, it is not necessary for each method in the interface to declare a `java.rmi.RemoteException` in its throws block. Exceptions that your application throws can be specific to that application and can extend `RuntimeException`.

Table 1-2 WebLogic RMI Ease of Use

Feature	WebLogic RMI
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 Manager	No Security Manager required. All WebLogic RMI services provided by WebLogic Server, which provides more sophisticated security options, such as SSL and ACLs. You can comment out the call to setSecurityManager() when converting RMI code to WebLogic RMI.
Inheritance	No requirement to extend UnicastRemoteObject, thus preserving your logical object hierarchy. Remote classes do not have to inherit from UnicastRemoteObject in order to inherit rmi.server package implementation. They can inherit classes from within your application hierarchy and yet retain the behavior of the rmi.server package.
Instrumentation and management	WebLogic Server, which hosts the RMI registry, provides a well-instrumented environment for development and deployment of distributed applications.

Table 1-3 WebLogic RMI Naming and Lookup

Feature	WebLogic RMI
Naming	Fully integrated with WebLogic JNDI. Applications can be partitioned into meaningful name spaces by using JNDI API or the WebLogic RMI registry interfaces. JNDI allows publication of RMI objects through enterprise naming services, such as LDAP or NDS.
Lookup	In URLs, use the standard rmi:// scheme, https://, iiop://, or http://, which tunnels WebLogic RMI requests over HTTP, making WebLogic RMI remote invocation available through firewalls.
Client-side invocation	Supports client-to-server, client-to-client, and server-to-client invocations. Operates within the well-defined WebLogic Server environment with optimized, multiplexed, asynchronous, and bidirectional client-server connections. Thus a client application can publish its objects through the registry, and other clients or servers can use the client-resident objects as they would any server-resident objects.

2 Programming Considerations

The following sections describe the WebLogic RMI features that you use to program RMI for use with WebLogic Server.

- WebLogic RMI Compiler
- WebLogic RMI Framework
- Dynamic Proxies in RMI
- Hot Code Generation
- WebLogic RMI Registry
- WebLogic RMI Implementation Features
- RMI and T3 Protocol

WebLogic RMI Compiler

The WebLogic RMI compiler (`weblogic.rmic`) generates dynamic proxies on the client-side for custom remote object interfaces and provides hot code generation for server-side objects. When `rmic` is run, the hot code generation feature generates bytecode that is dynamically created at runtime, when the RMI object is deployed.

Note: You only need to explicitly run `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.

The dynamic proxy class is the serializable class that is passed to the client. Hot code generation is the RMI feature that produces the bytecode is a server-side class that processes requests from the dynamic proxy on the client. The implementation for the class is bound to a name in the RMI registry in WebLogic Server.

A client acquires the proxy for the class by looking up the class in the 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. 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.

Dynamic Proxies and Bytecode

In previous versions of WebLogic Server, pre 6.1, running `rmic` generated stubs on the client and skeleton code on the server-side. Now, rmic generates an XML deployment descriptor which is loaded at runtime. Instead of a stub, the client uses a dynamic proxy to communicate with remote objects. A skeleton class is created on the fly in memory. 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 `rmic` on those objects. This will generate the necessary proxies and bytecode that enable the deployed RMI object.

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 Programming Enterprise JavaBeans for instructions on using `weblogic.ejbc`.

Rerunning either `weblogic.rmic` using 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

.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 tables list `java weblogic.rmic` options. Enter these options after `java weblogic.rmic` and before the name of the remote class.

Table 2-1 WebLogic RMI Compiler Options

Option	Description
`-callRouter <callRouterClass>`	Only for use 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 indicates that the current load algorithm should be used.
`-clusterable`	Marks the service as clusterable (can be hosted by multiple servers in a WebLogic cluster). Each hosting object, or replica, is bound into the naming service under a common name. When the service proxy 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.
`-commentary`	Emits commentary
`-dispatchPolicy <queueName>`	Specifies a configured execute queue that the service should use for obtaining execute threads in WebLogic Server. See Using Execute Queues to Control Thread Usage for more information.
`-help`	Prints a description of the options
`-idl`	Generates IDLs for remote interfaces
`-idlOverwrite`	Overwrites existing IDL files

Option	Description
`-idlVerbose`	Displays verbose information for IDL information
`-idlStrict`	Generates IDL according to OMG standard
`-idlNoFactories`	Prevents generation of factory methods for value types
`-idlDirectory <idlDirectory>`	Specifies the directory where IDL files will be created (Default = current directory)
`-iiop`	Generates IIOP proxy from servers
`-iiopDirectory`	Specifies the directory where IIOP proxy classes are written
`-keepgenerated`	Allows you to keep the source of generated proxy classes and bytecode when you run the WebLogic RMI compiler.
`-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.
`-methodsAreIdempotent`	Only for use in conduction with `-clusterable`. Indicates that the methods on this class are idem potent. This allows the proxy to attempt recovery form 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 proxy only retries on failures that are guaranteed to have occurred before the remote method was invoked.
`-nomangednames`	Causes the compiler to produce proxies specific to the remote class.
`-replicaListRefreshInterval <seconds>`	Only for use in conjunction with `-clusterable`. Specifies the minimum time to wait between attempts to refresh the replica list from the cluster (Default = 180 seconds).
`-stickToFirstServer`	Only for use in conjunction with `-clusterable`. Enables “sticky” load balancing. The server chosen for servicing the first request is used for all subsequent requests.

Option	Description
`-version`	Prints version information

Table 2-2 Cluster-Specific WebLogic RMI Compiler Options

Option	Description
`-callRouter <callRouterClass>`	Only for use 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` indicates that the current load algorithm should be used.
`-clusterable`	Marks the service as clusterable (can be hosted by multiple servers in a WebLogic cluster). Each hosting object, or replica, is bound into the naming service under a common name. When the service proxy 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. Load algorithm name may only be used in conjunction with `-clusterable`. Specifies a service-specific algorithm that will be used by the proxy to handle fail-over and load balancing. If this argument is unspecified, the default load balancing algorithm is specified in the Administration Console. For example, to specify weight-based load balancing: `$ java weblogic.rmic -clusterable -loadAlgorithm=weight-based`

Option	Description
`-methodsAreIdempotent`	May only be used in conjunction with `-clusterable`. Indicates to the proxy that it can attempt retries after fail-over even if it might result in executing the same method multiple times. If this flag isn't present, methods for this proxy are not considered idem potent. The exceptions that are handled by this are described in Exceptions Used for fail-over.
`-replicaListRefreshInterval <seconds>`	Only for use in conjunction with `-clusterable`. Specifies the minimum time to wait between attempts to refresh the replica list from the cluster (Default = 180 seconds).
`-stickToFirstServer`	Only for use in conjunction with `-clusterable`. Enables “sticky” load balancing. The server chosen for servicing the first request is used for all subsequent requests.

Replicating Stubs in a Cluster

You can also generate stubs that are *not* replicated in the cluster; these are known as "pinned" services, because after they are registered they will be available only from the host with which they are registered and will not provide transparent fail-over or load balancing.

If you use `weblogic.rmic` to compile an RMI object using the clustering option and then deploy the object on two nodes (A and B) of a three node server cluster A, B, and C) with replicating binding on all three nodes you get the same view from each server. When you do a JNDI lookup on all three nodes, you get the same stub and when you make method calls, the server performs load balancing between the first two nodes

Therefore, if you compile RMI objects with the clusterable option and bind them to a JNDI tree with replicate bindings set to false, when you do a JNDI lookup you get the following results:

- On Server A, you get a stub that points to Sever A
- On Server B, you get a stub that points to Server B
- On Server C, you get a NameNotFoundException

If you make a remote call to Server A fails and it fails because the server is no longer available, the clusterable stub does a re-lookup and depending on where the call is routed, one of the following can occur:

- On Server B, you get a stub that points to Server B
- On Server C, you get a NameNotFoundException

If your RMI object is non-clusterable and you bind it to a JNDI tree with replicate bindings set to `false`, when you do a JNDI lookup you get pinned stubs and there is no failover. 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 fail-over to another server.

If your RMI object is non-clusterable and you bind it to a JNDI tree with replicate bindings set to `true`, this will fail because the object is non-clusterable and only one server can provide a non-clusterable service in a cluster.

WebLogic RMI Framework

WebLogic RMI is divided between a client and server framework. The client runtime 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, the client must be connected 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 will only be reachable as long as the server that carried out the bind is reachable.

Additional WebLogic RMI Compiler Features

Other features of the WebLogic RMI compiler include:

- 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.

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, dynamic proxies are

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 WebLogic RMI Compiler with Proxies

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.

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.

WebLogic RMI Registry

WebLogic Server hosts the RMI registry and provides server infrastructure for RMI clients. The overhead for RMI registry and server communications is minimal, because registry traffic is multiplexed over the same connection as JDBC and other kinds of traffic. Clients use a single socket for RMI; scaling for RMI clients is linear in the WebLogic Server environment.

The WebLogic RMI registry is created when WebLogic Server starts up, and calls to create new registries simply locate the existing registry. Objects that have been bound in the registry can be accessed with a variety of client protocols, including the standard rmi://, as well as http://, or https://. In fact, all of the naming services use JNDI.

WebLogic RMI Implementation Features

In general, functional equivalents of all methods in the `java.rmi` package are provided in WebLogic RMI, except for those methods in the `RMIClassLoader` and the method `java.rmi.server.RemoteServer.getClientHost()`.

All other interfaces, exceptions, and classes are supported in WebLogic RMI. The following sections note particular implementations that may be of interest.

JNDI

Use Java Naming and Directory Interface (JNDI) as the preferred mechanism for naming objects in WebLogic RMI. The JNDI is an application programming interface (API) that provides naming services to Java applications. JNDI is an integral component of Sun Microsystems Inc.'s Java 2 Enterprise Edition (J2EE) technology. A naming service associates names with objects and finds objects based on their given names. (The RMI registry is an example of a naming service.)

Using JNDI with RMI allows you to make distributed programming more efficient. However, you should be aware of the number of round trips between remote client and the server. Repeated JNDI lookups between the client and server may cause performance problems

rmi.RMISecurityManager

`rmi.RMISecurityManager` is implemented as a non-final class with all public methods in WebLogic RMI, and, unlike the restrictive JavaSoft reference implementation, is entirely permissive. Security in WebLogic RMI is an integrated part of the larger WebLogic environment, for which there is support for SSL (Secure Socket Layer) and ACLs (Access Control Lists).

rmi.registry.LocateRegistry

`rmi.registry.LocateRegistry` is implemented as a final class with all public methods. However, a call to `LocateRegistry.createRegistry(int port)` does not create a collocated registry, but rather attempts to connect to the server-side instance that implements JNDI, for which host and port are designated by attributes. In WebLogic RMI, a call to this method allows the client to find the JNDI tree on WebLogic Server.

Note: You can use protocols other than the default (`rmi`) as well, and provide the scheme, host, and port as a URL, as shown here:

```
LocateRegistry.getRegistry(https://localhost:7002);
```

This example locates a WebLogic Server registry on the local host at port 7002, using a standard SSL protocol.

rmi.server Classes

`rmi.server.LogStream` diverges from the JavaSoft reference implementation in that the `write(byte[])` method logs messages through the WebLogic Server log file.

`rmi.server.RemoteObject` is implemented in WebLogic RMI to preserve the type equivalence of `UnicastRemoteObject`, but the functionality is provided by the WebLogic RMI base class `proxy`.

`rmi.server.RemoteServer` is implemented as the abstract super-class of `rmi.server.UnicastRemoteObject` and all public methods are supported in WebLogic RMI with the exception of `getClientHost()`.

`rmi.server.UnicastRemoteObject` is implemented as the base class for remote objects, and all the methods in this class are implemented in terms of the WebLogic RMI base class `Proxy`. This allows the proxy to override non-final `Object` methods and equate these to the implementation without making any requirements on the implementation.

In WebLogic RMI, all method parameters are pass-by-value, *unless* the invoking object resides in the same Java Virtual Machine (JVM) as the RMI object. In this scenario, method parameters are pass-by-reference.

Note: WebLogic RMI does not support uploading classes from the client. In other words, any classes passed to a remote object must be available within the server's `CLASSPATH`.

setSecurityManager

The `setSecurityManager()` method is provided in WebLogic RMI for compilation compatibility only. No security is associated with it, because WebLogic RMI depends on the more general security model within WebLogic Server. If, however, you *do* set a security manager, you can set only one. Before setting a security manager, you should test to see if one has already been set; if you try to set another, your program will throw an exception. Here is an example:

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

Unused Classes

```
System.setSecurityManager(new RMISecurityManager());
```

The following classes are implemented but unused in WebLogic RMI:

- `rmi.dgc.Lease`
- `rmi.dgc.VMID`
- `rmi.server.ObjID`
- `rmi.server.Operation`
- `rmi.server.RMIClassLoader`
- `rmi.server.RMISocketFactory`
- `rmi.server.UID`

RMI and T3 Protocol

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.

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.

3 Implementing WebLogic RMI

The following sections describe the WebLogic RMI API:

- Overview of the WebLogic RMI API
- Procedures for Implementing WebLogic RMI

Overview of the WebLogic RMI API

Several packages are shipped with WebLogic Server as part of WebLogic RMI. The public API includes:

- WebLogic implementation of the RMI base classes
- Registry
- Server packages
- WebLogic RMI compiler
- Supporting classes that are not part of the public API

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.

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 methods 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.

As in the JavaSoft reference implementation of RMI, the `java.rmi.Naming` class is an important one. It includes methods for binding, unbinding, and rebinding names to remote objects in the registry. It also includes a `lookup()` method to give a client access to a named remote object in the registry.

In addition, WebLogic JNDI provides naming and lookup services. WebLogic RMI supports naming and lookup in JNDI.

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

- Full Code Examples

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 compilation will 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. This allows you to retain a class hierarchy that makes sense for your application.

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.

The string name excepted by the RMI registry has the following syntax:

```
rmi://hostname:port/remoteObjectName
```

The hostname and port identify the machine and port on which the RMI registry is running and the remoteObjectName is the remote object’s string name. The hostname, port, and the prefix, rmi: are optional. If you do not specify a hostname, then WebLogic Server defaults to the local host. If you do not specify a port, then WebLogic Server uses 1099. If you do not specify the remoteObjectName, then the object being named is the RMI registry itself.

For more information, see the RMI specification.

Below is an example of a `main()` method for the `HelloImpl` class. This registers the `HelloImpl` object under the name `MultiHelloServer` 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);

      Naming.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

To run the WebLogic RMI compilern (`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 the WebLogic RMI compiler if you want to keep the generated wource if creating stubs and skeleton classes. For a listing of the available WebLogic RMI compiler options, see ".WebLogic RMI Compiler Options" on page 2-3.

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.

Full Code Examples

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();
      }
    }
  }
```