Java™ Remote Method Invocation Specification

Java™ Remote Method Invocation (RMI) is a distributed object model for the Java language that retains the semantics of the Java object model, making distributed objects easy to implement and to use. The system combines aspects of the Modula-3 Network Objects system and Spring’s subcontract and includes some novel features made possible by Java. The RMI system is easily extensible and maintainable.

Revision 1.4, JDK 1.1 FCS, February 10, 1997
Sun Microsystems, Inc. hereby grants to you a fully-paid, nonexclusive, nontransferable, perpetual, worldwide limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice this specification. This license allows and is limited to the creation and distribution of clean room implementations of this specification that (i) include a complete implementation of the current version of this specification without subsetting or supersetting, (ii) implement all the interfaces and functionality of the standard java.* packages as defined by SUN, without subsetting or supersetting, (iii) do not add any additional packages, classes or methods to the java.* packages (iv) pass all test suites relating to the most recent published version of this specification that are available from SUN six (6) months prior to any beta release of the clean room implementation or upgrade thereto, (v) do not derive from SUN source code or binary materials, and (vi) do not include any SUN binary materials without an appropriate and separate license from SUN.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, JavaBeans, JDK, Java, HotJava, the Java Coffee Cup logo, Java WorkShop, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, TooTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK® is a registered trademark of Novell, Inc.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

For further information on Intellectual Property matters contact Sun Legal Department:

Trademarks, Jan O'Dell at 415-786-8191
Patents at 415-336-0069
# Table of Contents

1 Introduction ....................................................... 1  
   1.1 Overview ............................................... 1  
   1.2 System Goals ......................................... 2  

2 Java Distributed Object Model ................................. 5  
   2.1 Definition of Terms .................................... 5  
   2.2 The Distributed and Nondistributed Models Contrasted 6  
   2.3 RMI Interfaces and Classes ............................ 7  
   2.4 Implementing a Remote Interface .................... 9  
   2.5 Type Equivalency of Remote Objects with Local Stub... 10  
   2.6 Parameter Passing in Remote Method Invocation ...... 10  
   2.7 Exception Handling in Remote Method Invocation ...... 11  
   2.8 Object Methods Overridden by the RemoteObject Class 11  
   2.9 The Semantics of Object Methods Declared final ...... 13  
   2.10 Locating Remote Objects ............................ 13
3 System Architecture .............................................. 15
  3.1 Overview ...................................................... 15
  3.2 Architectural Overview ...................................... 16
  3.3 The Stub/Skeleton Layer .................................... 18
  3.4 The Remote Reference Layer ................................. 19
  3.5 The Transport Layer ......................................... 20
  3.6 Thread Usage in Remote Method Invocations ............. 21
  3.7 Garbage Collection of Remote Objects ..................... 21
  3.8 Dynamic Class Loading ..................................... 22
  3.9 Security ...................................................... 26
  3.10 Configuration Scenarios .................................... 27
  3.11 RMI Through Firewalls Via Proxies ....................... 29
4 Client Interfaces .................................................. 33
  4.1 The Remote Interface ........................................ 33
  4.2 The RemoteException Class ............................... 33
  4.3 The Naming Class ........................................... 34
5 Server Interfaces .................................................. 37
  5.1 The RemoteObject Class .................................... 38
  5.2 The RemoteServer Class .................................... 38
  5.3 The UnicastRemoteObject Class ............................ 39
  5.4 The Unreferenced Interface ................................. 40
  5.5 The RMISecurityManager Class ............................ 41
  5.6 The RMIClassLoader Class .................................. 44
  5.7 The LoaderHandler Interface ............................... 45
9.3 RMI’s Use of Object Serialization Protocol ............... 71
9.4 RMI’s Use of HTTP POST Protocol ..................... 72
9.5 Application Specific Values for RMI .................... 72
9.6 RMI’s Multiplexing Protocol .............................. 73
A Exceptions In RMI ............................................ 81
   A.1 Exceptions During Remote Object Export ............... 82
   A.2 Exceptions During RMI Call ............................. 83
   A.3 Exceptions or Errors During Return .................... 83
   A.4 Naming Exceptions ....................................... 84
   A.5 Other Exceptions ........................................ 85
B Properties In RMI .............................................. 87
   B.1 Server Properties ....................................... 88
   B.2 Other Properties ....................................... 89
Introduction

Topics:
- Overview
- System Goals

1.1 Overview

Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate. For a basic communication mechanism, the Java™ language supports sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone.

An alternative to sockets is Remote Procedure Call (RPC), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR.

RPC, however, does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation,
distributed object systems require remote method invocation or RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

The Java remote method invocation system described in this specification has been specifically designed to operate in the Java environment. While other RMI systems can be adapted to handle Java objects, these systems fall short of seamless integration with the Java system due to their interoperability requirement with other languages. For example, CORBA presumes a heterogeneous, multilanguage environment and thus must have a language-neutral object model. In contrast, the Java language’s RMI system assumes the homogeneous environment of the Java Virtual Machine, and the system can therefore take advantage of the Java object model whenever possible.

1.2 System Goals

The goals for supporting distributed objects in the Java language are:

• Support seamless remote invocation on objects in different virtual machines.
• Support callbacks from servers to applets.
• Integrate the distributed object model into the Java language in a natural way while retaining most of the Java language’s object semantics.
• Make differences between the distributed object model and local Java object model apparent.
• Make writing reliable distributed applications as simple as possible.
• Preserve the safety provided by the Java runtime environment.

Underlying all these goals is a general requirement that the RMI model be both simple (easy to use) and natural (fits well in the language).

In addition, the RMI system should allow extensions such as garbage collection of remote objects, server replication, and the activation of persistent objects to service an invocation. These extensions should be transparent to the client and add minimal implementation requirements on the part of the servers that use them. To support these extensions, the system should also support:

• Several invocation mechanisms; for example simple invocation to a single object or invocation to an object replicated at multiple locations. The system should also be extensible to other invocation paradigms.
• Various reference semantics for remote objects; for example live (nonpersistent) references, persistent references, and lazy activation.
• The safe Java environment provided by security managers and class loaders.
• Distributed garbage collection of active objects.
• Capability of supporting multiple transports.

The first two chapters in this specification describe the distributed object model for the Java language and the system architecture. The remaining chapters describe the RMI client and server visible APIs which are part of JDK 1.1.
2.1 Definition of Terms

In the Java distributed object model, a *remote object* is one whose methods can be invoked from another Java Virtual Machine, potentially on a different host. An object of this type is described by one or more *remote interfaces*, which are Java interfaces that declare the methods of the remote object.
Remote method invocation (RMI) is the action of invoking a method of a remote interface on a remote object. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

2.2 The Distributed and Nondistributed Models Contrasted

The Java distributed object model is similar to the Java object model in the following ways:

- A reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote).
- A remote object can be cast to any of the set of remote interfaces supported by the implementation using the built-in Java syntax for casting.
- The built-in Java `instanceof` operator can be used to test the remote interfaces supported by a remote object.

The Java distributed object model differs from the Java object model in these ways:

- Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces.
- Nonremote arguments to, and results from, a remote method invocation are passed by copy rather than by reference. This is because references to objects are only useful within a single virtual machine.
- A remote object is passed by reference, not by copying the actual remote implementation.
- The semantics of some of the methods defined by class `Object` are specialized for remote objects.
- Since the failure modes of invoking remote objects are inherently more complicated than the failure modes of invoking local objects, clients must deal with additional exceptions that can occur during a remote method invocation.
2.3 RMI Interfaces and Classes

The interfaces and classes that are responsible for specifying the remote behavior of the RMI system are defined in the `java.rmi` and the `java.rmi.server` packages. The following figure shows the relationship between these interfaces and classes:

![Diagram showing the relationship between interfaces and classes](image)

### 2.3.1 The Remote Interface

All remote interfaces extend, either directly or indirectly, the interface `java.rmi.Remote`. The `Remote` interface defines no methods, as shown here:

```java
public interface Remote {
}
```

For example, the following code fragment defines a remote interface for a bank account that contains methods that deposit to the account, get the account balance, and withdraw from the account:

```java
public interface BankAccount
    extends Remote
{
    public void deposit (float amount)
        throws java.rmi.RemoteException;
    public void withdraw (float amount)
        throws OverdrawnException, java.rmi.RemoteException;
    public float balance()
        throws java.rmi.RemoteException;
}
```
The methods in a remote interface must be defined as follows:

- Each method must declare `java.rmi.RemoteException` in its throws clause, in addition to any application-specific exceptions.
- A remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

### 2.3.2 The `RemoteException` Class

The `java.rmi.RemoteException` class is the superclass of all exceptions that can be thrown by the RMI runtime. To ensure the robustness of applications using the RMI system, each method declared in a remote interface must specify `java.rmi.RemoteException` in its throws clause.

`java.rmi.RemoteException` is thrown when a remote method invocation fails (for example when the network fails or the server for the call cannot be reached). This allows the application making the remote invocation to determine how best to deal with the remote exception.

### 2.3.3 The `RemoteObject` Class and its Subclasses

RMI server functions are provided by `java.rmi.server.RemoteObject` and its subclasses, `java.rmi.server.RemoteServer` and `java.rmi.server.UnicastRemoteObject`:

- The `java.rmi.server.RemoteObject` class provides the remote semantics of `Object` by implementing methods for `hashCode`, `equals`, and `toString`.
- The functions needed to create objects and export them (make them available remotely) are provided abstractly by `java.rmi.server.RemoteServer` and concretely by its subclass(es). The subclass identifies the semantics of the remote reference, for example whether the server is a single object or is a replicated object requiring communications with multiple locations.
- The `java.rmi.server.UnicastRemoteObject` class defines a singleton (unicast) remote object whose references are valid only while the server process is alive.
2.4 Implementing a Remote Interface

The general rules for a class that implements a remote interface are as follows:

- The class usually extends `java.rmi.server.UnicastRemoteObject`, thereby inheriting the remote behavior provided by the classes `java.rmi.server.RemoteObject` and `java.rmi.server.RemoteServer`.
- The class can implement any number of remote interfaces.
- The class can extend another remote implementation class.
- The class can define methods that do not appear in the remote interface, but those methods can only be used locally and are not available remotely.

For example, the following code fragment defines the `BankAccountImpl` class, which implements the `BankAccount` remote interface and which extends the `java.rmi.server.UnicastRemoteObject` class:

```java
package my_package;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class BankAccountImpl extends UnicastRemoteObject implements BankAccount {
    public void deposit (float amount) throws RemoteException {
        ...
    }
    public void withdraw (float amount) throws OverdrawnException, RemoteException {
        ...
    }
    public float balance() throws RemoteException {
        ...
    }
}
```

Note that if necessary, a class that implements a remote interface can extend some other class besides `java.rmi.server.UnicastRemoteObject`. However, the implementation class must then assume the responsibility for the correct remote semantics of the `hashCode`, `equals`, and `toString` methods inherited from the `Object` class.
2.5 Type Equivalency of Remote Objects with Local Stub

In the distributed object model, clients interact with stub (surrogate) objects that have exactly the same set of remote interfaces defined by the remote object’s class; the stub class does not include the nonremote portions of the class hierarchy that constitutes the object’s type graph. This is because the stub class is generated from the most refined implementation class that implements one or more remote interfaces. For example, if C extends B and B extends A, but only B implements a remote interface, then a stub is generated from B, not C.

Because the stub implements the same set of remote interfaces as the remote object’s class, the stub has, from the point of view of the Java system, the same type as the remote portions of the server object’s type graph. A client, therefore, can make use of the built-in Java operations to check a remote object’s type and to cast from one remote interface to another.

Stubs are generated using the rmic compiler.

2.6 Parameter Passing in Remote Method Invocation

An argument to, or a return value from, a remote object can be any Java type that is serializable. This includes Java primitive types, remote Java objects, and nonremote Java objects that implement the java.io.Serializable interface. For more details on how to make classes serializable, see the Java “Object Serialization Specification.” For applets, if the class of an argument or return value is not available locally, it is loaded dynamically via the AppletClassLoader. For applications, these classes are loaded by the class loader that loaded the application; this is either the default class loader (which uses the local class path) or the RMIClassLoader (which uses the server’s codebase).

Some classes may disallow their being passed (by not being serializable), for example for security reasons. In this case the remote method invocation will fail with an exception.

2.6.1 Passing Nonremote Objects

A nonremote object, that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation, is passed by copy.
That is, when a nonremote object appears in a remote method invocation, the content of the nonremote object is copied before invoking the call on the remote object. By default, only the nonstatic and nontransient fields are copied. Similarly, when a nonremote object is returned from a remote method invocation, a new object is created in the calling virtual machine.

### 2.6.2 Passing Remote Objects

When passing a remote object as a parameter, the stub for the remote object is passed. A remote object passed as a parameter can only implement remote interfaces.

### 2.7 Exception Handling in Remote Method Invocation

Since remote methods include `java.rmi.RemoteException` in their signature, the caller must be prepared to handle those exceptions in addition to other application specific exceptions. When a `java.rmi.RemoteException` is thrown during a remote method invocation, the client may have little or no information on the outcome of the call — whether a failure happened before, during, or after the call completed. Therefore, remote interfaces and the calling methods declared in those interfaces should be designed with these failure semantics in mind.

### 2.8 Object Methods Overridden by the RemoteObject Class

The default implementations in the `Object` class for the `equals`, `hashCode`, and `toString` methods are not appropriate for remote objects. Therefore, the `java.rmi.server.RemoteObject` class provides implementations for these methods that have semantics more appropriate for remote objects. In this way, all objects that need to be available remotely can extend `java.rmi.server.RemoteObject` (typically indirectly via `java.rmi.server.UnicastRemoteObject`).

#### 2.8.1 equals and hashCode

In order for a remote object to be used as a key in a hash table, the methods `equals` and `hashCode` are overridden by the `java.rmi.server.RemoteObject` class:
• The `java.rmi.server.RemoteObject` class's implementation of the `equals` method determines whether two object references are equal, not whether the contents of the two objects are equal. This is because determining equality based on content requires a remote method invocation, and the signature of `equals` does not allow a remote exception to be thrown.

• The `java.rmi.server.RemoteObject` class's implementation of the `hashCode` method returns the same value for all remote references that refer to the same underlying remote object (because references to the same object are considered equal).

2.8.2 toString

The `toString` method is defined to return a string which represents the reference of the object. The contents of the string is specific to the reference type. The current implementation for singleton (unicast) objects includes information about the object specific to the transport layer (such as host name and port number) and an object identifier; references to replicated objects would contain more information.

2.8.3 clone

Objects are only cloneable using the Java language's default mechanism if they support the `java.lang.Cloneable` interface. Remote objects do not implement this interface, but do implement the `clone` method so that if subclasses need to implement `Cloneable` the remote classes will function correctly.

Client stubs are declared final and do not implement `clone`. Cloning a stub is therefore a local operation and cannot be used by clients to create a new remote object.

2.8.4 finalize

Remote object implementations (subclasses of `RemoteObject`) can use `finalize` to perform their own cleanup as necessary. For example, `finalize` can be used to deactivate an object server.
2.9 The Semantics of Object Methods Declared final

The following methods are declared final by the Object class and cannot be overridden:

- getClass
- notify
- notifyAll
- wait

The default implementation for getClass is appropriate for all Java objects, local or remote; the method needs no special implementation for remote objects. When used on a remote object, the getClass method reports the exact type of the generated stub object. Note that this type reflects only the remote interfaces implemented by the object, not its local interfaces.

The wait and notify methods of Object deal with waiting and notification in the context of the Java language’s threading model. While use of these methods for remote objects does not break the Java threading model, these methods do not have the same semantics as they do for local Java objects. Specifically, using these methods operates on the client’s local reference to the remote object (the stub), not the actual object at the remote site.

2.10 Locating Remote Objects

A simple bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based methods of the class java.rmi.Naming.

For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is usually obtained as a return value in a method call. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts. The java.rmi.Naming class provides Uniform Resource Locator (URL) based methods to look up, bind, rebind, unbind, and list the name-object pairings maintained on a particular host and port.

Here’s an example, (without exception handling) of how to bind and look up remote objects:
BankAccount acct = new BankAcctImpl();
String url = "rmi://java.Sun.COM/account";
// bind url to remote object
java.rmi.Naming.bind(url, acct);
...
// lookup account
acct = (BankAccount)java.rmi.Naming.lookup(url);
System Architecture

Topics:
- Overview
- Architectural Overview
- The Stub/Skeleton Layer
- The Remote Reference Layer
- The Transport Layer
- Thread Usage in Remote Method Invocations
- Garbage Collection of Remote Objects
- Dynamic Class Loading
- Security
- Configuration Scenarios
- RMI Through Firewalls Via Proxies

3.1 Overview

The RMI system consists of three layers: the stub/skeleton layer, the remote reference layer, and the transport layer. The boundary at each layer is defined by a specific interface and protocol; each layer, therefore, is independent of the next and can be replaced by an alternate implementation without affecting the
other layers in the system. For example, the current transport implementation is TCP-based (using Java sockets), but a transport based on UDP could be substituted.

To accomplish transparent transmission of objects from one address space to another, the technique of object serialization (designed specifically for the Java language) is used. Object serialization is described in this chapter only with regard to its use for marshaling primitives and objects. For complete details, see the Object Serialization Specification.

Another technique, called dynamic stub loading, is used to support client-side stubs which implement the same set of remote interfaces as a remote object itself. This technique, used when a stub of the exact type is not already available to the client, allows a client to use the Java language’s built-in operators for casting and type-checking.

### 3.2 Architectural Overview

The RMI system consists of three layers:

- The stub/skeleton layer — client-side stubs (proxies) and server-side skeletons
- The remote reference layer — remote reference behavior (such as invocation to a single object or to a replicated object)
- The transport layer — connection set up and management and remote object tracking

The application layer sits on top of the RMI system. The relationship between the layers is shown in the following figure.
A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote object as a conduit to the remote object. A client-held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote object and forwards invocation requests to that server object via the remote reference layer. Stubs are generated using the rmic compiler.

The remote reference layer is responsible for carrying out the semantics of the invocation. For example, the remote reference layer is responsible for determining whether the server is a single object or is a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics—whether the server is a single object or is a replicated object requiring communications with its replicas.

Also handled by the remote reference layer are the reference semantics for the server. The remote reference layer, for example, abstracts the different ways of referring to objects that are implemented in (a) servers that are always running on some machine, and (b) servers that are run only when some method invocation is made on them (activation). At the layers above the remote reference layer, these differences are not seen.

The transport layer is responsible for connection setup, connection management, and keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport’s address space.
In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to occur before handing off the request to the server-side skeleton. The skeleton for a remote object makes an up call to the remote object implementation which carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer, and transport on the server side, and then up through the transport, remote reference layer, and stub on the client side.

### 3.3 The Stub/Skeleton Layer

The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. This layer does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of marshal streams. Marshal streams employ a mechanism called object serialization which enables Java objects to be transmitted between address spaces. Objects transmitted using the object serialization system are passed by copy to the remote address space, unless they are remote objects, in which case they are passed by reference.

A stub for a remote object is the client-side proxy for the remote object. Such a stub implements all the interfaces that are supported by the remote object implementation. A client-side stub is responsible for:

- Initiating a call to the remote object (by calling the remote reference layer).
- Marshaling arguments to a marshal stream (obtained from the remote reference layer).
- Informing the remote reference layer that the call should be invoked.
- Unmarshaling the return value or exception from a marshal stream.
- Informing the remote reference layer that the call is complete.

A skeleton for a remote object is a server-side entity that contains a method which dispatches calls to the actual remote object implementation. The skeleton is responsible for:

- Unmarshaling arguments from the marshal stream.
- Making the up-call to the actual remote object implementation.
• Marshaling the return value of the call or an exception (if one occurred) onto the marshal stream.

The appropriate stub and skeleton classes are determined at run time and are dynamically loaded as needed, as described in Section 3.8, “Dynamic Class Loading”. Stubs and skeletons are generated using the rmic compiler.

3.4 The Remote Reference Layer

The remote reference layer deals with the lower-level transport interface. This layer is also responsible for carrying out a specific remote reference protocol which is independent of the client stubs and server skeletons.

Each remote object implementation chooses its own remote reference subclass that operates on its behalf. Various invocation protocols can be carried out at this layer. Examples are:

• Unicast point-to-point invocation.
• Invocation to replicated object groups.
• Support for a specific replication strategy.
• Support for a persistent reference to the remote object (enabling activation of the remote object).
• Reconnection strategies (if remote object becomes inaccessible).

The remote reference layer has two cooperating components: the client-side and the server-side components. The client-side component contains information specific to the remote server (or servers, if the remote reference is to a replicated object) and communicates via the transport to the server-side component. During each method invocation, the client and server-side components perform the specific remote reference semantics. For example, if a remote object is part of a replicated object, the client-side component can forward the invocation to each replica rather than just a single remote object.

In a corresponding manner, the server-side component implements the specific remote reference semantics prior to delivering a remote method invocation to the skeleton. This component, for example, would handle ensuring atomic multicast delivery by communicating with other servers in a replica group (note that multicast delivery is not part of the JDK 1.1 release of RMI).
The remote reference layer transmits data to the transport layer via the abstraction of a stream-oriented connection. The transport takes care of the implementation details of connections. Although connections present a streams-based interface, a connectionless transport can be implemented beneath the abstraction.

3.5 The Transport Layer

In general, the transport layer of the RMI system is responsible for:

- Setting up connections to remote address spaces.
- Managing connections.
- Monitoring connection “liveness.”
- Listening for incoming calls.
- Maintaining a table of remote objects that reside in the address space.
- Setting up a connection for an incoming call.
- Locating the dispatcher for the target of the remote call and passing the connection to this dispatcher.

The concrete representation of a remote object reference consists of an endpoint and an object identifier. This representation is called a live reference. Given a live reference for a remote object, a transport can use the endpoint to set up a connection to the address space in which the remote object resides. On the server side, the transport uses the object identifier to look up the target of the remote call.

The transport for the RMI system consists of four basic abstractions:

- An endpoint is the abstraction used to denote an address space or Java virtual machine. In the implementation, an endpoint can be mapped to its transport. That is, given an endpoint, a specific transport instance can be obtained.
- A channel is the abstraction for a conduit between two address spaces. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel.
- A connection is the abstraction for transferring data (performing input/output).
The transport abstraction manages channels. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces (the local address space and a remote address space). Given an endpoint to a remote address space, a transport sets up a channel to that address space. The transport abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system.

A transport defines what the concrete representation of an endpoint is, so multiple transport implementations may exist. The design and implementation also supports multiple transports per address space, so both TCP and UDP can be supported in the same virtual machine. Note that the RMI transport interfaces are only available to the virtual machine implementation and are not available directly to the application.

3.6 Thread Usage in Remote Method Invocations

A method dispatched by the RMI runtime to a remote object implementation (a server) may or may not execute in a separate thread. Some calls originating from the same client virtual machine will execute in the same thread; some will execute in different threads. Calls originating from different client virtual machines will execute in different threads. Other than this last case of different client virtual machines, the RMI runtime makes no guarantees with respect to mapping remote object invocations to threads.

3.7 Garbage Collection of Remote Objects

In a distributed system, just as in the local system, it is desirable to automatically delete those remote objects that are no longer referenced by any client. This frees the programmer from needing to keep track of the remote objects clients so that it can terminate appropriately. RMI uses a reference-counting garbage collection algorithm similar to Modula-3’s Network Objects. (See “Network Objects” by Birrell, Nelson, and Owicki, Digital Equipment Corporation Systems Research Center Technical Report 115, 1994.)

To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each Java virtual machine. When a live reference enters a Java virtual machine, its reference count is incremented. The first reference to an object sends a “referenced” message to the server for the
object. As live references are found to be unreferenced in the local virtual machine, their finalization decrements the count. When the last reference has been discarded, an unreferenced message is sent to the server. Many subtleties exist in the protocol; most of these are related to maintaining the ordering of referenced and unreferenced messages in order to ensure that the object is not prematurely collected.

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine's garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine's garbage collector in the usual ways by holding normal or weak references to objects. As in the normal object life-cycle finalize will be called after the garbage collector determines that no more references to the object exist.

As long as a local reference to a remote object exists, it cannot be garbage-collected and it can be passed in remote calls or returned to clients. Passing a remote object adds the identifier for the virtual machine to which it was passed to the referenced set. A remote object needing unreferenced notification must implement the java.rmi.server.Unreferenced interface. When those references no longer exist, the unreferenced method will be invoked.

unreferenced is called when the set of references is found to be empty so it might be called more than once. Remote objects are only collected when no more references, either local or remote, still exist.

Note that if a network partition exists between a client and a remote server object, it is possible that premature collection of the remote object will occur (since the transport might believe that the client crashed). Because of the possibility of premature collection, remote references cannot guarantee referential integrity; in other words, it is always possible that a remote reference may in fact not refer to an existing object. An attempt to use such a reference will generate a RemoteException which must be handled by the application.

### 3.8 Dynamic Class Loading

In RPC (remote procedure call) systems, client-side stub code must be generated and linked into a client before a remote procedure call can be done. This code can be either statically linked into the client or linked in at runtime.
via dynamic linking with libraries available locally or over a network file system. In the case of either static or dynamic linking, the specific code to handle an RPC must be available to the client machine in compiled form.

RMI generalizes this technique, using a mechanism called dynamic class loading to load at runtime (in the Java language’s architecture neutral bytecode format) the classes required to handle method invocations on a remote object. These classes are:

- The classes of remote objects and their interfaces.
- The stub and skeleton classes that serve as proxies for remote objects. (Stubs and skeletons are created using the rmic compiler.)
- Other classes used directly in an RMI-based application, such as parameters to, or return values from, remote method invocations.

This section describes:

- How the RMI runtime chooses a class loader and the location from which to load classes.
- How to force the downloading over the net of all the classes for a Java application.

In addition to class loaders, dynamic class loading employs two other mechanisms: the object serialization system to transmit classes over the wire, and a security manager to check the classes that are loaded. The object serialization system is discussed in the Object Serialization Specification. Security issues are discussed in Section 3.9, “Security.”

3.8.1 How a Class Loader is Chosen

In Java, the class loader that initially loads a Java class is subsequently used to load all the interfaces and classes that are used directly in the class:

- The AppletClassLoader is used to download a Java applet over the net from the location specified by the codebase attribute on the web page that contains the <applet> tag. All classes used directly in the applet are subsequently loaded by the AppletClassLoader.
• The default class loader is used to load a class (whose `main` method is run by using the `java` command) from the local CLASSPATH. All classes used directly in that class are subsequently loaded by the default class loader from the local CLASSPATH.

• The RMIClassLoader is used to load those classes not directly used by the client or server application: the stubs and skeletons of remote objects, and extended classes of arguments and return values to RMI calls. The RMIClassLoader looks for these classes in the following locations, in the order listed:
  a. The local CLASSPATH. Classes are always loaded locally if they exist locally.
  b. For objects (both remote and nonremote) passed as parameters or return values, the URL encoded in the marshal stream that contains the serialized object is used to locate the class for the object.
  c. For stubs and skeletons of remote objects created in the local virtual machine, the URL specified by the local `java.rmi.server.codebase` property is used.

For objects passed as parameters or return values (the second case above), the URL that is encoded in the stream for an object's class is determined as follows:

• If the class was loaded by a class loader (other than the default classloader), the URL of that class loader is used.

• otherwise, if defined, the `java.rmi.server.codebase` URL is used.

Thus, if a class was loaded from CLASSPATH, the codebase URL will be used to annotate that class in the stream if that class is used in an RMI call.

The application can be configured with the property `java.rmi.server.useCodebaseOnly`, which disables the loading of classes from network hosts and forces classes to be loaded only from the locally defined codebase. If the required class cannot be loaded, the method invocation will fail with an exception.
3.8.2 Bootstrapping the Client

For the RMI runtime to be able to download all the classes and interfaces needed by a client application, a bootstrapping client program is required which forces the use of a class loader (such as RMI’s class loader) instead of the default class loader. The bootstrapping program needs to:

- Create an instance of the RMISecurityManager or user-defined security manager.
- Use the method RMIClassLoader.loadClass to load the class file for the client. The class name cannot be mentioned explicitly in the code, but must instead be a string or a command line argument. Otherwise, the default class loader will try to load the client class file from the local CLASSPATH.
- Use the newInstance method to create an instance of the client and cast it to Runnable. Thus, the client must implement the java.lang.Runnable interface. The Runnable interface provides a well-defined interface for starting a thread of execution.
- Start the client by calling the run method (of the Runnable interface).

For example:

```java
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;

public class LoadClient
{
    public static void main()
    {
        System.setSecurityManager(new RMISecurityManager());
        try {
            Class cl = RMIClassLoader.loadClass("myclient");
            Runnable client = (Runnable)cl.newInstance();
            client.run();
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

In order for this code to work, you need to specify the java.rmi.server.codebase property when you run the bootstrapping program so that the loadClass method will use this URL to load the class. For example:
Instead of relying on the property, you can supply your own URL:

```java
Class cl = RMIClassLoader.loadClass(url, "myclient");
```

Once the client is started and has control, all classes needed by the client will be loaded from the specified URL. This bootstrapping technique is exactly the same technique Java uses to force the AppletClassLoader to download the same classes used in an applet.

Without this bootstrapping technique, all the classes directly referenced in the client code must be available through the local CLASSPATH on the client, and the only Java classes that can be loaded by the RMIClassLoader over the network are classes that are not referred to directly in the client program; these classes are stubs, skeletons, and the extended classes of arguments and return values to remote method invocations.

### 3.9 Security

In Java, when a class loader loads classes from the local CLASSPATH, those classes are considered trustworthy and are not restricted by a security manager. However, when the RMIClassLoader attempts to load classes from the network, there must be a security manager in place or an exception is thrown.

The security manager must be started as the first action of a Java program so that it can regulate subsequent actions. The security manager ensures that loaded classes adhere to the standard Java safety guarantees, for example that classes are loaded from “trusted” sources (such as the applet host) and do not attempt to access sensitive functions. A complete description of the restrictions imposed by security managers can be found in the documentation for the AppletSecurity class and the RMISecurityManager class.

Applets are always subject to the restrictions imposed by the AppletSecurity class. This security manager ensures that classes are loaded only from the applet host or its designated codebase hosts. This requires that applet developers install the appropriate classes on the applet host.

Applications must either define their own security manager or use the restrictive RMISecurityManager. If no security manager is in place, an application cannot load classes from network sources.
A client or server program is usually implemented by classes loaded from the local system and therefore is not subject to the restrictions of the security manager. If however, the client program itself is downloaded from the network using the technique described in Section 3.8.2, “Bootstrapping the Client”, then the client program is subject to the restrictions of the security manager.

**Note** – Once a class is loaded by the RMIClassLoader, any classes used directly by that class are also loaded by the RMIClassLoader and thus are subject to the security manager restrictions.

Even if a security manager is in place, setting the property `java.rmi.server.useCodebaseOnly` to true prevents the downloading of a class from the URL embedded in the stream with a serialized object (classes can still be loaded from the locally-defined `java.rmi.server.codebase`). The `java.rmi.server.useCodebaseOnly` property can be specified on both the client and the server, but is not applicable for applets.

If an application defines its own security manager which disallows the creation of a class loader, classes will be loaded using the default `Class.forName` mechanism. Thus, a server may define its own policies via the security manager and class loader, and the RMI system will operate within those policies.

**Note** – The `java.lang.SecurityManager` abstract class, from which all security managers are extended, does not regulate resource consumption. Therefore, the current `RMISecurityManager` has no mechanisms available to prevent classes loaded from abusing resources. As new security manager mechanisms are developed, RMI will use them.

### 3.10 Configuration Scenarios

The RMI system supports many different scenarios. Servers can be configured in an open or closed fashion. Applets can use RMI to invoke methods on objects supported on servers. If an applet creates and passes a remote object to the server, the server can use RMI to make a callback to the remote object. Java applications can use RMI either in client-server mode or from peer to peer. This section highlights the issues surrounding these configurations.
3.10.1 Servers

The typical closed-system scenario has the server configured to load no classes. The services it provides are defined by remote interfaces that are all local to the server machine. The server has no security manager and will not load classes even if clients send along the URL. If clients send remote objects for which the server does not have stub classes, those method invocations will fail when the request is unmarshaled, and the client will receive an exception.

The more open server system will define its `java.rmi.server.codebase` so that classes for the remote objects it exports can be loaded by clients, and so that the server can load classes when needed for remote objects supplied by clients. The server will have both a security manager and RMI class loader which protect the server. A somewhat more cautious server can use the property `java.rmi.server.useCodebaseOnly` to disable the loading of classes from client-supplied URLs.

3.10.2 Applets

Typically, the classes needed will be supplied by an HTTP server or by an FTP server as referenced in URL’s embedded in the HTML page containing the applet. The RMI-based service(s) used by the applet must be on the server from which the applet was downloaded, because an applet can only make network connections to the host from which it was loaded.

For example, the normal applet scenario uses a single host for the HTTP server providing the HTML page, the applet code, the RMI services, and the bootstrap Registry. In this scenario, all the stub, skeleton, and supporting classes are loaded from the HTTP server. All of the remote objects provided by the RMI service and passed to the applet (which may pass them back to the server) will be for classes that the RMI service already knows about. In this case, the RMI service is very secure because it loads no classes from the network.
3.10.3 Applications

Applications written in the Java language, unlike applets, can connect to any host; so Java applications have more options for configuring the sources of classes and where RMI based services run. Typically, a single HTTP server will be used to supply remote classes, while the RMI-based applications themselves are distributed around the network on servers or running on user’s desktops.

If an application is loaded locally, then the classes used directly in that program must also be available locally. In this scenario, the only classes that can be downloaded from a network source are the classes of remote interfaces, stub classes, and the extended classes of arguments and return values to remote method invocations.

If an application is not loaded from a local directory, but is loaded from a network source using the bootstrapping mechanism described in Section 3.8.2, “Bootstrapping the Client”, then all classes used by the application can be downloaded from the same network source.

To enable downloading from a network source, each remote object server must be configured with the java.rmi.server.codebase property which specifies where application classes and generated stubs/skeletons reside. When the codebase property is specified, the RMI system embeds the URL of a class in the serialized form of the class.

Even if a serialized object’s class is annotated with the URL from which the class can be downloaded, a client or peer will still load classes locally if they are available.

3.11 RMI Through Firewalls Via Proxies

The RMI transport layer normally attempts to open direct sockets to hosts on the Internet. Many intranets, however, have firewalls which do not allow this. The default RMI transport, therefore, provides two alternate HTTP-based mechanisms which enable a client behind a firewall to invoke a method on a remote object which resides outside the firewall.
3.11.1 How an RMI Call is Packaged within the HTTP Protocol

To get outside a firewall, the transport layer embeds an RMI call within the firewall-trusted HTTP protocol. The RMI call data is sent outside as the body of an HTTP POST request, and the return information is sent back in the body of the HTTP response. The transport layer will formulate the POST request in one of two ways:

1. If the firewall proxy will forward an HTTP request directed to an arbitrary port on the host machine, then it is forwarded directly to the port on which the RMI server is listening. The default RMI transport layer on the target machine is listening with a server socket that is capable of understanding and decoding RMI calls inside POST requests.

2. If the firewall proxy will only forward HTTP requests directed to certain well-known HTTP ports, then the call will be forwarded to the HTTP server listening on port 80 of the host machine, and a CGI script will be executed to forward the call to the target RMI server port on the same machine.

3.11.2 The Default Socket Factory

The RMI transport extends the java.rmi.server.RMISocketFactory class to provide a default implementation of a socket factory which is the resource-provider for client and server sockets. This default socket factory creates sockets that transparently provide the firewall tunnelling mechanism as follows:

- Client sockets automatically attempt HTTP connections to hosts that cannot be contacted with a direct socket.
- Server sockets automatically detect if a newly-accepted connection is an HTTP POST request, and if so, return a socket that will expose only the body of the request to the transport and format its output as an HTTP response.

Client-side sockets, with this default behavior, are provided by the factory's java.rmi.server.RMISocketFactory.createSocket method. Server-side sockets with this default behavior are provided by the factory's java.rmi.server.RMISocketFactory.createServerSocket method.
3.11.3 Configuring the Client

There is no special configuration necessary to enable the client to send RMI calls through a firewall.

The client can, however, disable the packaging of RMI calls as HTTP requests by setting the `java.rmi.server.disableHttp` property to equal the boolean value `true`.

3.11.4 Configuring the Server

**Note** – The host name should not be specified as the host’s IP address, because some firewall proxies will not forward to such a host name.

1. In order for a client outside the server host’s domain to be able to invoke methods on a server’s remote objects, the client must be able to find the server. To do this, the remote references that the server exports must contain the fully-qualified name of the server host.

   Depending on the server’s platform and network environment, this information may or may not be available to the Java virtual machine on which the server is running. If it is not available, the host’s fully qualified name must be specified with the property `java.rmi.server.hostname` when starting the server.

   For example, use this command to start the RMI server class `ServerImpl` on the machine `chatsubo.javasoft.com`:

   ```
   java -Djava.rmi.server.hostname=chatsubo.javasoft.com ServerImpl
   ```

2. If the server will not support RMI clients behind firewalls that can forward to arbitrary ports, use this configuration:

   a. An HTTP server is listening on port 80.

   b. A CGI script is located at the aliased URL path `/cgi-bin/java-rmi`. This script:
      - Invokes the local Java interpreter to execute a class internal to the transport layer which forwards the request to the appropriate RMI server port.
• Defines properties in the Java virtual machine with the same names and values as the CGI 1.0 defined environment variables.

An example script is supplied in the RMI distribution for the Solaris and Windows 32 operating systems. Note that the script must specify the complete path to the java interpreter on the server machine.

3.11.5 Performance Issues and Limitations

Calls transmitted via HTTP requests are at least an order of magnitude slower than those sent through direct sockets, without taking proxy forwarding delays into consideration.

Because HTTP requests can only be initiated in one direction through a firewall, a client cannot export its own remote objects outside the firewall, because a host outside the firewall cannot initiate a method invocation back on the client.