# Introduction

***1.1* Background**

Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate. For a basic communication mechanism, the JavaTM programming 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 platform's remote method invocation system described in this specification has been specifically designed to operate in the Java application environment. The Java programming language's RMI system assumes the homogeneous environment of the Java virtual machine (JVM), and the system can therefore take advantage of the Java platform's object model whenever possible.

## *1.2* System Goals

The goals for supporting distributed objects in the Java programming 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 programming language in a natural way while retaining most of the Java programming language's object semantics
* Make differences between the distributed object model and local Java platform's object model apparent
* Make writing reliable distributed applications as simple as possible
* Preserve the type-safety provided by the Java platform's runtime environment
* Support various reference semantics for remote objects; for example live (nonpersistent) references, persistent references, and lazy activation
* Maintain the safe environment of the Java platform provided by security managers and class loaders

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).

The first two chapters in this specification describe the distributed object model for the Java programming language and the system overview. The remaining chapters describe the RMI client and server visible APIs which are part of the Java SE platform.

# Distributed Object Model

***2.1* Distributed Object Applications**

RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates a number of remote objects, makes references to those remote objects accessible, and waits for clients to invoke methods on those remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to:

* Locate remote objects

Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, the rmiregistry, or the application can pass and return remote object references as part of its normal operation.

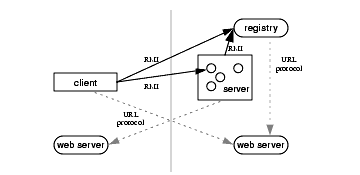
* Communicate with remote objects

Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard method invocation.

* Load class bytecodes for objects that are passed as parameters or return values

Because RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object's code as well as transmitting its data.

The illustration below depicts an RMI distributed application that uses the registry to obtain references to a remote object. The server calls the registry to associate a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load bytecodes of classes written in the Java programming language, from server to client and from client to server, for objects when needed. RMI can load class bytecodes using any URL protocol (e.g., HTTP, FTP, file, etc.) that is supported by the Java platform.



## *2.2* Definition of Terms

In the Java SE platform's 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 interfaces written in the Java programming language 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.

***2.3* The Distributed and Nondistributed Models Contrasted**

The Java SE platform's distributed object model is similar to the Java SE platform's object model in the following ways:

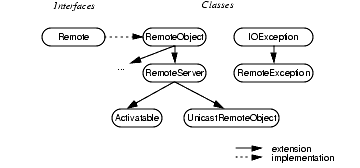
* A reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote).
* A remote object can be cast to any of the set of remote interfaces supported by the implementation using the syntax for casting built into the Java programming language.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

The Java SE platform's distributed object model differs from the Java SE platform's object model in these ways:

* Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces.
* Non-remote 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 java.lang.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.

## *2.4* Overview of 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 package hierarchy. The following figure shows the relationship between several of these interfaces and classes:



### *2.4.1* The java.rmi.Remote Interface

In RMI, a remote interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine. A remote interface must satisfy the following requirements:

* A remote interface must at least extend, either directly or indirectly, the interface java.rmi.Remote.
* Each method declaration in a remote interface or its super-interfaces must satisfy the requirements of a remote method declaration as follows:
  + A remote method declaration must include the exception java.rmi.RemoteException (or one of its superclasses such as java.io.IOException orjava.lang.Exception) in its throws clause, in addition to any application-specific exceptions (note that application specific exceptions do not have to extendjava.rmi.RemoteException).
  + In a remote method declaration, a remote object declared as a parameter or return value (either declared directly in the parameter list or embedded within a non-remote object in a parameter) must be declared as the remote interface, not the implementation class of that interface.

The interface java.rmi.Remote is a marker interface that defines no methods:

public interface Remote {}

A remote interface must at least extend the interface java.rmi.Remote (or another remote interface that extends java.rmi.Remote). However, a remote interface may extend a non-remote interface under the following condition:

* A remote interface may also extend another non-remote interface, as long as all of the methods (if any) of the extended interface satisfy the requirements of a remote method declaration.

For example, the following interface BankAccount defines a remote interface for accessing a bank account. It contains remote methods to deposit to the account, to get the account balance, and to withdraw from the account:

public interface BankAccount extends java.rmi.Remote {

public void deposit(float amount)

throws java.rmi.RemoteException;

public void withdraw(float amount)

throws OverdrawnException, java.rmi.RemoteException;

public float getBalance()

throws java.rmi.RemoteException;

}

The next example shows a valid remote interface Beta that extends a non-remote interface Alpha, which has remote methods, and the interface java.rmi.Remote:

public interface Alpha {

public final String okay = "constants are okay too";

public Object foo(Object obj)

throws java.rmi.RemoteException;

public void bar() throws java.io.IOException;

public int baz() throws java.lang.Exception;

}

public interface Beta extends Alpha, java.rmi.Remote {

public void ping() throws java.rmi.RemoteException;

}

### *2.4.2* The RemoteException Class

The java.rmi.RemoteException class is the superclass of exceptions thrown by the RMI runtime during a remote method invocation. To ensure the robustness of applications using the RMI system, each remote method declared in a remote interface must specify java.rmi.RemoteException (or one of its superclasses such as java.io.IOExceptionor java.lang.Exception) in its throws clause.

The exception java.rmi.RemoteException is thrown when a remote method invocation fails for some reason. Some reasons for remote method invocation failure include:

* Communication failure (the remote server is unreachable or is refusing connections; the connection is closed by the server, etc.)
* Failure during parameter or return value marshalling or unmarshalling
* Protocol errors

The class RemoteException is a checked exception (one that must be handled by the caller of a remote method and is checked by the compiler), not a RuntimeException.

### *2.4.3* 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 andjava.rmi.activation.Activatable.

* The class java.rmi.server.RemoteObject provides implementations for the java.lang.Object methods, hashCode, equals, and toString that are sensible for remote objects.
* The methods needed to create remote objects and export them (make them available to remote clients) are provided by the classes UnicastRemoteObject andActivatable. The subclass identifies the semantics of the remote reference, for example whether the server is a simple remote object or is an activatable remote object (one that executes when invoked).
* The java.rmi.server.UnicastRemoteObject class defines a singleton (unicast) remote object whose references are valid only while the server process is alive.
* The class java.rmi.activation.Activatable is an abstract class that defines an activatable remote object that starts executing when its remote methods are invoked and can shut itself down when necessary.

***2.5* 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 andjava.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 class BankAcctImpl implements the BankAccount remote interface and extends the java.rmi.server.UnicastRemoteObject class:

package mypackage;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class BankAccountImpl

extends UnicastRemoteObject

implements BankAccount

{

private float balance = 0.0;

public BankAccountImpl(float initialBalance)

throws RemoteException

{

balance = initialBalance;

}

public void deposit(float amount) throws RemoteException {

...

}

public void withdraw(float amount) throws OverdrawnException,

RemoteException {

...

}

public float getBalance() 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 exporting the object (taken care of by the UnicastRemoteObject constructor) and for implementing (if needed) the correct remote semantics of the hashCode, equals, and toString methods inherited from the java.lang.Object class.

## *2.6* Parameter Passing in Remote Method Invocation

An argument to, or a return value from, a remote object can be any object that is serializable. This includes primitive types, remote objects, and non-remote objects that implement the java.io.Serializable interface. For more details on how to make classes serializable, see the "Java Object Serialization Specification." Classes, for parameters or return values, that are not available locally are downloaded dynamically by the RMI system. See the section on "[Dynamic Class Loading](http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-arch5.html)" for more information on how RMI downloads parameter and return value classes when reading parameters, return values and exceptions.

### *2.6.1* Passing Non-remote Objects

A non-remote 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, the object is serialized using the object serialization mechanism of the Java SE platform.

So, when a non-remote object is passed as an argument or return value in a remote method invocation, the content of the non-remote object is copied before invoking the call on the remote object.

When a non-remote 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 an exported remote object as a parameter or return value in a remote method call, the stub for that remote object is passed instead. Remote objects that are not exported will not be replaced with a stub instance. A remote object passed as a parameter can only implement remote interfaces.

### *2.6.3* Referential Integrity

If two references to an object are passed from one JVM to another JVM in parameters (or in the return value) in a single remote method call and those references refer to the same object in the sending JVM, those references will refer to a single copy of the object in the receiving JVM. More generally stated: within a single remote method call, the RMI system maintains referential integrity among the objects passed as parameters or as a return value in the call.

### *2.6.4* Class Annotation

When an object is sent from one JVM to another in a remote method call, the RMI system annotates the class descriptor in the call stream with information (the URL) of the class so that the class can be loaded at the receiver. It is a requirement that classes be downloaded on demand during remote method invocation.

### *2.6.5* Parameter Transmission

Parameters in an RMI call are written to a stream that is a subclass of the class java.io.ObjectOutputStream in order to serialize the parameters to the destination of the remote call. The ObjectOutputStream subclass overrides the replaceObject method to replace each exported remote object with its corresponding stub instance. Parameters that are objects are written to the stream using the ObjectOutputStream's writeObject method. The ObjectOutputStream calls the replaceObject method for each object written to the stream via the writeObject method (that includes objects referenced by those objects that are written). The replaceObject method of RMI's subclass ofObjectOutputStream returns the following:

* If the object passed to replaceObject is an instance of java.rmi.Remote and that object is exported to the RMI runtime, then it returns the stub for the remote object. If the object is an instance of java.rmi.Remote and the object is not exported to the RMI runtime, then replaceObject returns the object itself. A stub for a remote object is obtained via a call to the method java.rmi.server.RemoteObject.toStub.
* If the object passed to replaceObject is not an instance of java.rmi.Remote, then the object is simply returned.

RMI's subclass of ObjectOutputStream also implements the annotateClass method that annotates the call stream with the location of the class so that it can be downloaded at the receiver. See the section "[Dynamic Class Loading](http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-arch5.html)" for more information on how annotateClass is used.

Since parameters are written to a single ObjectOutputStream, references that refer to the same object at the caller will refer to the same copy of the object at the receiver. At the receiver, parameters are read by a single ObjectInputStream.

Any other default behavior of ObjectOutputStream for writing objects (and similarly ObjectInputStream for reading objects) is maintained in parameter passing. For example, the calling of writeReplace when writing objects and readResolve when reading objects is honored by RMI's parameter marshal and unmarshal streams.

In a similar manner to parameter passing in RMI as described above, a return value (or exception) is written to a subclass of ObjectOutputStream and has the same replacement behavior as parameter transmission.

## *2.7* 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 classjava.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 parameter or 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.

# RMI System Overview

***3.1* Stubs and Skeletons**

RMI uses a standard mechanism (employed in RPC systems) for communicating with remote objects: *stubs* and *skeletons*. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements.

When a stub's method is invoked, it does the following:

* initiates a connection with the remote JVM containing the remote object,
* marshals (writes and transmits) the parameters to the remote JVM,
* waits for the result of the method invocation,
* unmarshals (reads) the return value or exception returned, and
* returns the value to the caller.

The stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote JVM, each remote object may have a corresponding skeleton (in Java 2 platform-only environments, skeletons are not required). The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

* unmarshals (reads) the parameters for the remote method,
* invokes the method on the actual remote object implementation, and
* marshals (writes and transmits) the result (return value or exception) to the caller.

In the Java 2 SDK, Standard Edition, v1.2 an additional stub protocol was introduced that eliminates the need for skeletons in Java 2 platform-only environments. Instead, generic code is used to carry out the duties performed by skeletons in JDK1.1. Stubs and skeletons are generated by the rmic compiler.

## *3.2* Thread Usage in Remote Method Invocations

A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads. Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe.

## *3.3* 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, the count is decremented. 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 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 thejava.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.

## *3.4* Dynamic Class Loading

RMI allows parameters, return values and exceptions passed in RMI calls to be any object that is serializable. RMI uses the object serialization mechanism to transmit data from one virtual machine to another and also annotates the call stream with the appropriate location information so that the class definition files can be loaded at the receiver.

When parameters and return values for a remote method invocation are unmarshalled to become live objects in the receiving JVM, class definitions are required for all of the types of objects in the stream. The unmarshalling process first attempts to resolve classes by name in its local class loading context (the context class loader of the current thread). RMI also provides a facility for dynamically loading the class definitions for the actual types of objects passed as parameters and return values for remote method invocations from network locations specified by the transmitting endpoint. This includes the dynamic downloading of remote stub classes corresponding to particular remote object implementation classes (and used to contain remote references) as well as any other type that is passed by value in RMI calls, such as the subclass of a declared parameter type, that is not already available in the class loading context of the unmarshalling side.

To support dynamic class loading, the RMI runtime uses special subclasses of java.io.ObjectOutputStream and java.io.ObjectInputStream for the marshal streams that it uses for marshalling and unmarshalling RMI parameters and return values. These subclasses respectively override the annotateClass method of ObjectOutputStream and theresolveClass method of ObjectInputStream to communicate information about where to locate class files containing the definitions for classes corresponding to the class descriptors in the stream.

For every class descriptor written to an RMI marshal stream, the annotateClass method adds to the stream the result of callingjava.rmi.server.RMIClassLoader.getClassAnnotation for the class object, which may be null or may be a String object representing the codebase URL path (a space-separated list of URLs) from which the remote endpoint should download the class definition file for the given class.

For every class descriptor read from an RMI marshal stream, the resolveClass method reads a single object from the stream. If the object is a String (and the value of thejava.rmi.server.useCodebaseOnly property is not true), then resolveClass returns the result of calling RMIClassLoader.loadClass with the annotated String object as the first parameter and the name of the desired class in the class descriptor as the second parameter. Otherwise, resolveClass returns the result of callingRMIClassLoader.loadClass with the name of the desired class as the only parameter.

See the section "[The RMIClassLoader Class](http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-server27.html)" for more details about class loading in RMI.

## *3.5* 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 that 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.

As described in this section, the HTTP-based mechanism that the RMI transport layer uses for RMI calls only applies to firewalls with HTTP proxy servers.

### *3.5.1* 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:

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

### *3.5.2* The Default Socket Factory

The RMI transport implementation includes an extension of the class java.rmi.server.RMISocketFactory, which is the default resource-provider for client and server sockets used to send and receive RMI calls; this default socket factory can be obtained via the java.rmi.server.RMISocketFactory.getDefaultSocketFactory method. This default socket factory creates sockets that transparently provide the firewall tunnelling mechanism as follows:

* Client sockets first attempt a direct socket connection. Client sockets automatically attempt HTTP connections to hosts that cannot be contacted with a direct socket if that direct socket connection results in either a java.net.NoRouteToHostException or a java.net.UnknownHostException being thrown. If a direct socket connection results in any other java.io.IOException being thrown, such as a java.net.ConnectException, the implementation may attempt an HTTP connection.
* 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.

### *3.5.3* Configuring the Client

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

### *3.5.4* 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.example.com:

java -Djava.rmi.server.hostname=chatsubo.example.com ServerImpl

1. If the server will not support RMI clients behind firewalls that can forward to arbitrary ports, use this configuration:
   1. An HTTP server is listening on port 80.
   2. A CGI script is located at the aliased URL path
   3. /cgi-bin/java-rmi.cgi

This script:

* + - Invokes the local interpreter for the Java programming language 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 interpreter for the Java programming language on the server machine.

### *3.5.5* Performance Issues and Limitations

Calls transmitted via HTTP requests are at least an order of magnitude slower that 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.

# Client Interfaces

When writing an applet or an application that uses remote objects, the programmer needs to be aware of the RMI system's client visible interfaces that are available in thejava.rmi package.

## *4.1* The Remote Interface

See the Remote API documentation, and for more details on how to define a remote interface see the section "[The java.rmi.Remote Interface](http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-objmodel5.html#3459)".

## *4.2* The RemoteException Class

See the RemoteException API documentation.

## *4.3* The Naming Class

See the Naming API documentation.

# Server Interfaces

The java.rmi.server package contains interfaces and classes typically used to implement remote objects.

## *5.1* The RemoteObject Class

See the RemoteObject API documentation.

## *5.2* The RemoteServer Class

See the RemoteServer API documentation.

## *5.3* The UnicastRemoteObject Class

The class java.rmi.server.UnicastRemoteObject provides support for creating and exporting remote objects. The class implements a remote server object with the following characteristics:

* References to such objects are valid only for, at most, the life of the process that creates the remote object.
* Communication with the remote object uses a TCP transport.
* Invocations, parameters, and results use a stream protocol for communicating between client and server.

package java.rmi.server;

public class UnicastRemoteObject extends RemoteServer {

protected UnicastRemoteObject()

throws java.rmi.RemoteException {...}

protected UnicastRemoteObject(int port)

throws java.rmi.RemoteException {...}

protected UnicastRemoteObject(int port,

RMIClientSocketFactory csf,

RMIServerSocketFactory ssf)

throws java.rmi.RemoteException {...}

public Object clone()

throws java.lang.CloneNotSupportedException {...}

public static RemoteStub exportObject(java.rmi.Remote obj)

throws java.rmi.RemoteException {...}

public static Remote exportObject(java.rmi.Remote obj, int port)

throws java.rmi.RemoteException {...}

public static Remote exportObject(Remote obj, int port,

RMIClientSocketFactory csf,

RMIServerSocketFactory ssf)

throws java.rmi.RemoteException {...}

public static boolean unexportObject(java.rmi.Remote obj,

boolean force)

throws java.rmi.NoSuchObjectException {...}

}

### *5.3.1* Constructing a New Remote Object

A remote object implementation (one that implements one or more remote interfaces) must be created and exported. Exporting a remote object makes that object available to accept incoming calls from clients. For a remote object implementation that is exported as a UnicastRemoteObject, the exporting involves listening on a TCP port (note that more than one remote object can accept incoming calls on the same port, so listening on a new port is not always necessary). A remote object implementation can extend the class UnicastRemoteObject to make use of its constructors that export the object, or it can extend some other class (or none at all) and export the object viaUnicastRemoteObject's exportObject methods.

The no argument constructor creates and exports a remote object on an anonymous (or arbitrary) port, chosen at runtime. The second form of the constructor takes a single argument, port, that specifies the port number on which the remote object accepts incoming calls. The third constructor creates and exports a remote object that accepts incoming calls on the specified port via a ServerSocket created from the RMIServerSocketFactory; clients will make connections to the remote object via Sockets supplied from the RMIClientSocketFactory.

### *5.3.2* Exporting an Implementation Not Extended From RemoteObject

An exportObject method (any of the forms) is used to export a simple peer-to-peer remote object that is not implemented by extending the UnicastRemoteObject class. The first form of the exportObject method takes a single parameter, obj, which is the remote object that will accept incoming RMI calls; this exportObject method exports the object on an anonymous (or arbitrary) port, chosen at runtime. The second exportObject method takes two parameters, both the remote object, obj, and port, the port number on which the remote object accepts incoming calls. The third exportObject method exports the object, obj, with the specified RMIClientSocketFactory, csf, andRMIServerSocketFactory, ssf, on the specified port.

The exportObject method returns a Remote stub which is the stub object for the remote object, obj, that is passed in place of the remote object in an RMI call.

### *5.3.3* Passing a UnicastRemoteObject in an RMI Call

As stated above, when an exported object of type UnicastRemoteObject is passed as a parameter or return value in an RMI call, the object is replaced by the remote object's stub. An exported remote object implementation remains in the virtual machine in which it was created and does not move (even by value) from that virtual machine. In other words, an exported remote object is passed by reference in an RMI call; exported remote object implementations cannot be passed by value.

### *5.3.4* Serializing a UnicastRemoteObject

Information contained in UnicastRemoteObject is transient and is not saved if an object of that type is written to a user-defined ObjectOutputStream (for example, if the object is written to a file using serialization). An object that is an instance of a user-defined subclass of UnicastRemoteObject, however, may have non-transient data that can be saved when the object is serialized.

When a UnicastRemoteObject is read from an ObjectInputStream using UnicastRemoteObject's readObject method, the remote object is automatically exported to the RMI runtime so that it may receive RMI calls. If exporting the object fails for some reason, deserializing the object will terminate with an exception.

### *5.3.5* Unexporting a UnicastRemoteObject

The unexportObject method makes the remote object, obj, unavailable for incoming calls. If the force parameter is true, the object is forcibly unexported even if there are pending calls to the remote object or the remote object still has calls in progress. If the force parameter is false, the object is only unexported if there are no pending or in-progress calls to the object. If the object is successfully unexported, the RMI runtime removes the object from its internal tables. Unexporting the object in this forcible manner may leave clients holding stale remote references to the remote object. This method throws java.rmi.NoSuchObjectException if the object was not previously exported to the RMI runtime.

### *5.3.6* The clone method

Objects are only clonable using the Java programming language's default mechanism if they support the java.lang.Cloneable interface. The classjava.rmi.server.UnicastRemoteObject does not implement this interface, but does implement the clone method so that if subclasses need to implement Cloneable, the remote object will be capable of being cloned properly. The clone method can be used by a subclass to create a cloned remote object with initially the same contents, but is exported to accept remote calls and is distinct from the original object.

## *5.4* The Unreferenced Interface

package java.rmi.server;

public interface Unreferenced {

public void unreferenced();

}

The java.rmi.server.Unreferenced interface allows a server object to receive notification that there are no clients holding remote references to it. The distributed garbage collection mechanism maintains for each remote object, the set of client virtual machines that hold references to that remote object. As long as some client holds a remote reference to the remote object, the RMI runtime keeps a local reference to the remote object. Each time the remote object's "reference" set becomes empty (meaning that the number of clients that reference the object becomes zero), the Unreferenced.unreferenced method is invoked (if that remote object implements the Unreferenced interface). A remote object is not required to support the Unreferenced interface.

As long as some local reference to the remote object exists, it may be passed in remote calls or returned to clients. The process that receives the reference is added to the reference set for the remote object. When the reference set becomes empty, the remote object's unreferenced method will be invoked. As such, the unreferenced method can be called more than once (each time the set is newly emptied). Remote objects are only collected when no more references, either local references or those held by clients, still exist.

## *5.5* The RMISecurityManager Class

See the RMISecurityManager API documentation.

## *5.6* The RMIClassLoader Class

See the RMIClassLoader API documentation.

## *5.7* The LoaderHandler Interface

See the LoaderHandler API documentation.

## *5.8* RMI Socket Factories

When the RMI runtime implementation needs instances of java.net.Socket and java.net.ServerSocket for its connections, instead of instantiating objects of those classes directly, it calls the createSocket and createServerSocket methods on the current RMISocketFactory object, returned by the static methodRMISocketFactory.getSocketFactory. This allows the application to have a hook to customize the type of sockets used by the RMI transport, such as alternate subclasses of the java.net.Socket and java.net.ServerSocket classes. The instance of RMISocketFactory to be used can be set once by trusted system code. In JDK1.1, this customization was limited to relatively global decisions about socket type, because the only parameters supplied to the factory's methods were host and port (forcreateSocket) and just port (for createServerSocket).

In the Java SE platform, the new interfaces RMIServerSocketFactory and RMIClientSocketFactory have been introduced to provide more flexible customization of what protocols are used to communicate with remote objects.

To allow applications using RMI to take advantage of these new socket factory interfaces, several new constructors and exportObject methods, that take the client and server socket factory as additional parameters, have been added to both UnicastRemoteObject and java.rmi.activation.Activatable.

Remote objects exported with either of the new constructors or exportObject methods (with RMIClientSocketFactory and RMIServerSocketFactory parameters) will be treated differently by the RMI runtime. For the lifetime of such a remote object, the runtime will use the custom RMIServerSocketFactory to create a ServerSocket to accept incoming calls to the remote object and use the custom RMIClientSocketFactory to create a Socket to connect clients to the remote object.

The implementation of RemoteRef and ServerRef used in the stubs and skeletons for remote objects exported with custom socket factories is UnicastRef2 andUnicastServerRef2, respectively. The wire representation of the UnicastRef2 type contains a different representation of the "endpoint" to contact than the UnicastRef type has (which used just a host name string in UTF format, following by an integer port number). For UnicastRef2, the endpoint's wire representation consists of a format byte specifying the contents of the rest of the endpoint's representation (to allow for future expansion of the endpoint representation) followed by data in the indicated format. Currently, the data may consist of a hostname in UTF format, a port number, and optionally (as specified by the endpoint format byte) the serialized representation of anRMIClientSocketFactory object that is used by clients to generate socket connections to remote object at this endpoint. The endpoint representation does not contain theRMIServerSocketFactory object that was specified when the remote object was exported.

When calls are made through references of the UnicastRef2 type, the runtime uses the createSocket method of the RMIClientSocketFactory object in the endpoint when creating sockets for connections to the referent remote object. Also, when the runtime makes DGC "dirty" and "clean" calls for a particular remote object, it must call the DGC on the remote JVM using a connection generated from the same RMIClientSocketFactory object as specified in the remote reference, and the DGC implementation on the server side should verify that this was done correctly.

Remote objects exported with the older constructor or method on UnicastRemoteObject that do not take custom socket factories as arguments will have RemoteRef andServerRef of type UnicastRef and UnicastServerRef as before and use the old wire representation for their endpoints, i.e. a host string in UTF format followed by an integer specifying the port number. This is so that RMI servers that do not use new 1.2 features will interoperate with older RMI clients.

### *5.8.1* The RMISocketFactory Class

The java.rmi.server.RMISocketFactory abstract class provides an interface for specifying how the transport should obtain sockets. Note that the class below uses Socketand ServerSocket from the java.net package.

package java.rmi.server;

public abstract class RMISocketFactory

implements RMIClientSocketFactory, RMIServerSocketFactory

{

public abstract Socket createSocket(String host, int port)

throws IOException;

  public abstract ServerSocket createServerSocket(int port)

throws IOException;

public static void setSocketFactory(RMISocketFactory fac)

throws IOException {...}

public static RMISocketFactory getSocketFactory() {...}

public static void setFailureHandler(RMIFailureHandler fh) {...}

public static RMIFailureHandler getFailureHandler() {...}

}

The static method setSocketFactory is used to set the socket factory from which RMI obtains sockets. The application may invoke this method with its ownRMISocketFactory instance only once. An application-defined implementation of RMISocketFactory could, for example, do preliminary filtering on the requested connection and throw exceptions, or return its own extension of the java.net.Socket or java.net.ServerSocket classes, such as ones that provide a secure communication channel. Note that the RMISocketFactory may only be set if the current security manager allows setting a socket factory; if setting the socket factory is disallowed, aSecurityException will be thrown.

The static method getSocketFactory returns the socket factory used by RMI. The method returns null if the socket factory is not set.

The transport layer invokes the createSocket and createServerSocket methods on the RMISocketFactory returned by the getSocketFactory method when the transport needs to create sockets. For example:

RMISocketFactory.getSocketFactory().createSocket(myhost, myport)

The method createSocket should create a client socket connected to the specified host and port. The method createServerSocket should create a server socket on the specified port.

The default transport's implementation of RMISocketFactory provides for transparent RMI through firewalls using HTTP as follows:

* On createSocket, the factory automatically attempts HTTP connections to hosts that cannot be contacted with a direct socket.
* On createServerSocket, the factory returns a server socket that automatically detects if a newly accepted connection is an HTTP POST request. If so, it returns a socket that will transparently expose only the body of the request to the transport and format its output as an HTTP response.

The method setFailureHandler sets the failure handler to be called by the RMI runtime if the creation of a server socket fails. The failure handler returns a boolean to indicate if retry should occur. The default failure handler returns false, meaning that by default recreation of sockets is not attempted by the runtime.

The method getFailureHandler returns the current handler for socket creation failure, or null if the failure handler is not set.

### *5.8.2* The RMIServerSocketFactory Interface

See the RMIServerSocketFactory API documentation.

### *5.8.3* The RMIClientSocketFactory Interface

See the RMIClientSocketFactory API documentation.

## *5.9* The RMIFailureHandler Interface

The java.rmi.server.RMIFailureHandler interface provides a method for specifying how the RMI runtime should respond when server socket creation fails (except during object export).

package java.rmi.server;

public interface RMIFailureHandler {

public boolean failure(Exception ex);

}

The failure method is invoked with the exception that prevented the RMI runtime from creating a java.net.ServerSocket. The method returns true if the runtime should attempt to retry and false otherwise.

Before this method can be invoked, a failure handler needs to be registered via the RMISocketFactory.setFailureHandler call. If the failure handler is not set, the RMI runtime attempts to re-create the ServerSocket after waiting for a short period of time.

Note that the RMIFailureHandler is not called when ServerSocket creation fails upon initial export of the object. The RMIFailureHandler will be called when there is an attempt to create a ServerSocket after a failed accept on that ServerSocket.

## *5.10* The LogStream Class

See the LogStream API documentation.

***5.11* Stub and Skeleton Compiler**

The rmic stub and skeleton compiler is used to compile the appropriate stubs and skeletons for a specific remote object implementation.

Please see the following URLs for further information on rmic:

* For the SolarisTM Operating System:

http://java.sun.com/products/jdk/1.6.0/docs/tooldocs/solaris/rmic.html

* For the Windows platform:

http://java.sun.com/products/jdk/1.6.0/docs/tooldocs/windows/rmic.html

# Registry Interfaces

The RMI system uses the java.rmi.registry.Registry interface and the java.rmi.registry.LocateRegistry class to provide a well-known bootstrap service for retrieving and registering objects by simple names.

A *registry* is a remote object that maps names to remote objects. Any server process can support its own registry or a single registry can be used for a host.

The methods of LocateRegistry are used to get a registry operating on a particular host or host and port. The methods of the java.rmi.Naming class makes calls to a remote object that implements the Registry interface using the appropriate LocateRegistry.getRegistry method.

## *6.1* The Registry Interface

See the Registry API documentation.