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..

Revision 1.50, JDK 1.2, October 1998

Table of Contents

Introduction 1

Topics:

- Background
- System Goals

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 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. 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 type-safety provided by the Java runtime environment.
- 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.

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 language and the system overview. The remaining chapters describe the RMI client and server visible APIs which are part of JDK 1.2.

Java Distributed Object Model 2

Topics:

- Distributed Object Applications
- Definition of Terms
- The Distributed and Nondistributed Models Contrasted
- Overview of RMI Interfaces and Classes
- Implementing a Remote Interface
- Parameter Passing in Remote Method Invocation
- Locating Remote Objects

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 applications gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mecahnism by which the server and the client communicate and pass information back and forth. Such an applications 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.

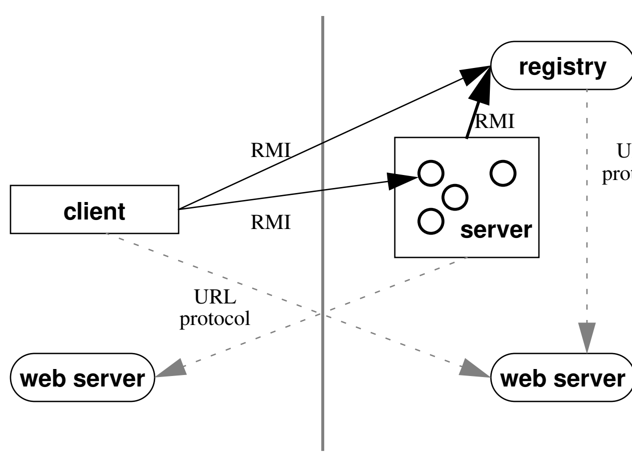
- Communicate with remote objects

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

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

  Because RMI allows a caller to pass pure Java 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 Java class bytecodes, 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 system.



2.2 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.

2.3 The Distributed and Nondistributed Models Contrasted

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

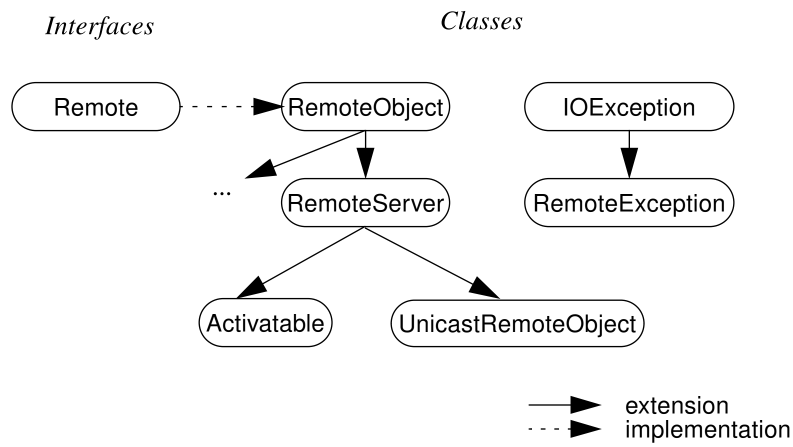
- A reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote).
- A remote object can be cast to any of the set of remote interfaces supported by the implementation using the built-in Java syntax for casting.
- The built-in Java `instanceof` 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.
- 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 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` or `java.lang.Exception`) in its throws clause, in addition to any application-specific exceptions (note that application specific exceptions do not have to extend `java.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.IOException` or `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` and `java.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` and `Activatable`. 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` 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 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 Java object that is *serializable*. This includes Java primitive types, remote Java objects, and non-remote Java 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" 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 Java Object Serialization mechanism.

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 a remote object as a parameter or return value in a remote method call, the stub for the remote object is passed. 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 VM to another VM in parameters (or in the return value) in a single remote method call and those references refer to the same object in the sending VM, those references will refer to a single copy of the object in the receiving VM. 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 VM 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 remote object with its corresponding stub class. 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 of `ObjectOutputStream` returns the following:

- if the object passed to `replaceObject` is an instance of `java.rmi.Remote`, then it returns the stub for the remote object. 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" 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 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 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

Topics:

- Stubs and Skeletons
- Thread Usage in Remote Method Invocations
- Garbage Collection of Remote Objects
- Dynamic Class Loading
- RMI Through Firewalls Via Proxies

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 reponsible 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 VM containing the remote object.
- marshals (writes and transmits) the parameters to the remote VM
- waits for the result of the method invocation
- unmarshals (reads) the return value or exception returned

- 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 VM, each remote object may have a corresponding skeleton (in JDK1.2-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
- marshals (writes and transmits) the result (return value or exception) to the caller

In JDK1.2 and additional stub protocol was introduced that eliminates the need for skeletons in JDK1.2-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 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.

3.4 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 VM, 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 override the `annotateClass` method of `ObjectOutputStream` and the `resolveClass` 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 calling `java.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 the `java.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 calling `RMIClassLoader.loadClass` with the name of the desired class as the only parameter.

See the section "The RMIClassLoader Class" for more details about classloading 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 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.

3.5.1 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:

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.

3.5.2 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.

3.5.3 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.

3.5.4 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.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.cgi
   ```

   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.

3.5.5 *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 4

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 the `java.rmi` package.

Topics:

- The Remote Interface
- The RemoteException Class
- The Naming Class

4.1 The Remote Interface

```
package java.rmi;
public interface Remote {}
```

The `java.rmi.Remote` interface serves to identify all remote interfaces; all remote objects must directly or indirectly implement this interface.

Implementation classes can implement any number of remote interfaces and can extend other remote implementation classes. RMI provides some convenience classes that remote object implementations can extend which facilitate remote object creation. These classes are `java.rmi.server.UnicastRemoteObject` and `java.rmi.activation.Activatable`.

For more details on how to define a remote interface see the section "The java.rmi.Remote Interface".

4.2 The RemoteException Class

The class `java.rmi.RemoteException` is the common superclass of a number of communication-related exceptions that may occur during the execution of a remote method call. Each method of a remote interface, an interface, must list `RemoteException` (or one of its superclasses such as `java.io.IOException` or `java.lang.Exception`) in its throws clause.

```
package java.rmi;
public class RemoteException extends java.io.IOException
{
    public Throwable detail;
    public RemoteException();
    public RemoteException(String s);
    public RemoteException(String s, Throwable ex);
    public String getMessage();
    public void printStackTrace();
    public void printStackTrace(java.io.PrintStream ps);
    public void printStackTrace(java.io.PrintWriter pw);
}
```

A `RemoteException` can be constructed with a detail message, *s*, and a nested exception, *ex* (a `Throwable`). Typically, the nested exception, *ex*, specified as a parameter in the third form of the constructor, is the underlying I/O exception that occurred during an RMI call.

The `getMessage` method returns the detail message of the exception, including the message from the nested exception (if any).

The `printStackTrace` methods are overridden from the class `java.lang.Throwable` to print out the stack trace of the nested exception.

4.3 The Naming Class

The `java.rmi.Naming` class provides methods for storing and obtaining references to remote objects in the remote object registry. The `Naming` class's methods take, as one of their arguments, a name that is URL formatted `java.lang.String` of the form:

```
//host:port/name
```

where *host* is the host (remote or local) where the registry is located, *port* is the port number on which the registry accepts calls, and where *name* is a simple string uninterpreted by the registry. Both *host* and *port* are optional. If *host* is omitted, the host defaults to the local host. If *port* is omitted, then the port defaults to 1099, the “well-known” port that RMI’s registry, `rmiregistry`, uses.

Binding a name for a remote object is associating or registering a name for a remote object that can be used at a later time to look up that remote object. A remote object can be associated with a name using the `Naming` class’s `bind` or `rebind` methods.

Once a remote object is registered (bound) with the RMI registry on the local host, callers on a remote (or local) host can lookup the remote object by name, obtain its reference, and then invoke remote methods on the object. A registry may be shared by all servers running on a host or an individual server process may create and use its own registry if desired (see `java.rmi.registry.LocateRegistry.createRegistry` method for details).

```
package java.rmi;
public final class Naming {
    public static Remote lookup(String url)
        throws NotBoundException, java.net.MalformedURLException,
        RemoteException;
    public static void bind(String url, Remote obj)
        throws AlreadyBoundException,
        java.net.MalformedURLException, RemoteException;
    public static void rebind(String url, Remote obj)
        throws RemoteException, java.net.MalformedURLException;
    public static void unbind(String url)
        throws RemoteException, NotBoundException,
        java.net.MalformedURLException;
    public static String[] list(String url)
        throws RemoteException, java.net.MalformedURLException;
}
```

The `lookup` method returns the remote object associated with the file portion of the name. The `NotBoundException` is thrown if the name has not been bound to an object.

The `bind` method binds the specified name to the remote object. It throws the `AlreadyBoundException` if the name is already bound to an object.

The `rebind` method always binds the name to the object even if the name is already bound. The old binding is lost.

The `unbind` method removes the binding between the name and the remote object. It will throw the `NotBoundException` if there was no binding.

The `list` method returns an array of `String` objects containing a snapshot of the URLs bound in the registry. Only the host and port information of the URL is needed to contact a registry for the list of its contents; thus, the “file” part of the URL is ignored.

Note – The `java.rmi.AccessException` may also be thrown as a result of any of these methods. The `AccessException` indicates that the caller does not have permission to execute the specific operation. For example, only clients that are local to the host on which the registry runs are permitted to execute the operations, `bind`, `rebind`, and `unbind`. A `lookup` operation, however can be invoked from any non-local client.

Server Interfaces 5

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

Topics:

5.1 The RemoteObject Class

The `java.rmi.server.RemoteObject` class implements the `java.lang.Object` behavior for remote objects. The `hashCode` and `equals` methods are implemented to allow remote object references to be stored in hashtables and compared. The `equals` method returns true if two `Remote` objects refer to the same remote object. It compares the remote object references of the remote objects.

The `toString` method returns a string describing the remote object. The contents and syntax of this string is implementation-specific and can vary.

All of the other methods of `java.lang.Object` retain their original implementations.

```
package java.rmi.server;
public abstract class RemoteObject
    implements java.rmi.Remote, java.io.Serializable
{
    protected transient RemoteRef ref;
    protected RemoteObject();
    protected RemoteObject(RemoteRef ref);
    public RemoteRef getRef();
    public static Remote toStub(java.rmi.Remote obj)
        throws java.rmi.NoSuchObjectException;
    public int hashCode();
    public boolean equals(Object obj);
    public String toString();
}
```

Since the `RemoteObject` class is abstract, it cannot be instantiated. Therefore, one of RemoteObject's constructors must be called from a subclass implementation. The first `RemoteObject` constructor creates a `RemoteObject` with a null remote reference. The second `RemoteObject` constructor creates a `RemoteObject` with the given remote reference, *ref*.

The `getRef` method returns the remote reference for the remote object.

The `toStub` method returns a stub for the remote object, *obj*, passes as a parameter. This operation is only valid after the remote object implementation has been exported. If the stub for the remote object could not be found, then the method throws `NoSuchObjectException`.

5.1.1 Object Methods Overridden by the RemoteObject Class

The default implementations in the `java.lang.Object` class for the `equals`, `hashCode`, and `toString` methods are not appropriate for remote objects. Therefore, the `RemoteObject` class provides implementations for these methods that have semantics more appropriate for remote objects.

equals and hashCode methods

In order for a remote object to be used as a key in a hash table, the methods `equals` and `hashCode` need to be overridden in the remote object implementation. These methods are overridden by the class `java.rmi.server.RemoteObject`:

- 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).

toString method

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

clone method

Objects are only clonable using the Java language's default mechanism if they support the `java.lang.Cloneable` interface. Stubs for remote objects generated by the `rmic` compiler are declared final and do not implement the `Cloneable` interface. Therefore, cloning a stub is not possible.

5.1.2 Serialized Form

The `RemoteObject` class implements the special (private) `writeObject` and `readObject` methods called by the object serialization mechanism to handle serializing data to a `java.io.ObjectOutputStream`. `RemoteObject`'s serialized form is written using the method:

```
private void writeObject(java.io.ObjectOutputStream out)
    throws java.io.IOException, java.lang.ClassNotFoundException;
```

- If `RemoteObject`'s remote reference field, *ref*, is null, then the method throws `java.rmi.MarshalException`.
- If the remote reference, *ref*, is non-null:
  - *ref*'s class is obtained via a call to its `getRefClass` method, which typically returns the unpackage qualified name of the remote reference's class. If the class name returned is non-null:
    - *ref*'s class name is written to the stream, *out*, in UTF.
    - *ref*'s `writeExternal` method is called passing it the stream, *out*, so that *ref* can write its external representation to the stream.
  - If the class name returned by `ref.getRefClass` is null:
    - an empty string in UTF is written to the stream *out*.
    - *ref* is serialized to the stream *out* (i.e., using writeObject).

A `RemoteObject`'s state is reconstructed from its serialized form using this method called by the `ObjectInputStream` during deserialization:

```
private void readObject(java.io.ObjectInputStream in)
    throws java.io.IOException, java.lang.ClassNotFoundException;
```

- First, the *ref*'s class name, a UTF string, is read from the stream *in*. If the class name is an empty string:
  - an object is read from the stream, and *ref* is initialized to that object (i.e., by a call to `in.readObject`)
- If the class name is a non-empty string:
  - The *ref*'s full class name is constructed by conatenating the value of the string `java.rmi.server.RemoteRef.packagePrefix` and "." with the class name read from the stream.
  - An instance of the ref's class is created (from the full class name).
  - The new instance (which becomes the *ref* field) reads its external form from the stream, *in*.

5.2 The RemoteServer Class

The `java.rmi.server.RemoteServer` class is the common superclass to the server implementation classes `java.rmi.server.UnicastRemoteObject` and `java.rmi.activation.Activatable`.

```
package java.rmi.server;
public abstract class RemoteServer extends RemoteObject {

    protected RemoteServer();
    protected RemoteServer(RemoteRef ref);

    public static String getClientHost()
        throws ServerNotActiveException;
    public static void setLog(java.io.OutputStream out);
    public static java.io.PrintStream getLog();
}
```

Since the `RemoteServer` class is abstract, it cannot be instantiated. Therefore, one of RemoteServer's constructors must be called from a subclass implementation. The first `RemoteServer` constructor creates a `RemoteServer` with a null remote reference. The second `RemoteServer` constructor creates a `RemoteServer` with the given remote reference, *ref*.

The `getClientHost` method allows an active method to determine the host that initiated the remote method active in the current thread. The exception `ServerNotActiveException` is thrown if no remote method is active in the current thread. The `setLog` method logs RMI calls to the specified output stream. If the output stream is null, call logging is turned off. The `getLog` method returns the stream for the RMI call log, so that application-specific information can be written to the call log in a synchronized manner.

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 implemention 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 via `UnicastRemoteObject`'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*, and `RMIServerSocketFactory`, *ssf*, on the specified *port*.

The object *must* be exported prior to the first time it is passed in an RMI call as either a parameter or return value, otherwise, a `java.rmi.server.StubNotFoundException` is thrown when a remote call is attempted in which an "unexported" remote object is passed as an argument or return value.

Once exported, the object can be passed as an argument in an RMI call or returned as the result of an RMI call.

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 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. A 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, a remote object is passed by reference in an RMI call; 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`, it 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 language's default mechanism if they support the `java.lang.Cloneable` interface. The class `java.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 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. When the "reference" set becomes empty, the `Unreferenced.unreferenced` method is invoked (if the server 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

```
package java.rmi;

public class RMISecurityManager extends java.lang.SecurityManager {

    public RMISecurityManager();
    public synchronized void checkPackageAccess(String pkg)
        throws RMISecurityException;
}
```

The `RMISecurityManager` provides the same security features as the java.lang.SecurityManager, but overrides the `checkPackageAcess` method.

In RMI applications, if no security manager has been set, stubs and classes can only be loaded from the local classpath. This ensures that the application is protected from code that is downloaded as a result of remote method invocations.

5.6 The RMIClassLoader Class

The java.rmi.server.RMIClassLoader class provides a set of public static utility methods for supporting network-based class loading in RMI. These methods are called by RMI's internal marshal streams to implement the dynamic class loading of types for RMI parameters and return values, but they also may be called directly by applications in order to mimic RMI's class loading behavior. The RMIClassLoader class has no publicly-accessible constructors and thus cannot be instantiated.

```
package java.rmi.server;

public class RMIClassLoader {
    public static String getClassAnnotation(Class cl);
    public static Object getSecurityContext(ClassLoader loader);
    public static Class loadClass(String name)
        throws java.net.MalformedURLException,
               ClassNotFoundException;
    public static Class loadClass(String codebase, String name)
        throws java.net.MalformedURLException,
              ClassNotFoundException;
    public static Class loadClass(URL codebase, String name)
        throws java.net.MalformedURLException,
              ClassNotFoundException;
}
```

The `getClassAnnotation` method returns a `String` representing the network codebase path that a remote endpoint should use for downloading the definition of the indicated class. The RMI runtime uses `String` objects returned by this method as the annotations for class descriptors in its marshal streams. The format of this codebase string is a path of codebase URL strings delimited by spaces.

The codebase string returned depends on the class loader of the supplied class:

- If the class loader is one of the following:
  - the "system class loader" (the class loader used to load classes in the application's "class path" and returned by the method `ClassLoader.getSystemClassLoader`),
  - a parent of the "system class loader" such as the class loader used for installed extensions,
  - or null (the "boot class loader" used to load VM classes),

then the value of the `java.rmi.server.codebase` property is returned, or null is returned if that property is not set.

- Otherwise, if the class loader is an instance of the class `java.net.URLClassLoader`, then the codebase string returned is a space-separated list of the external forms of the URLs returned by invoking the `getURLs` methods on the class loader. If the `URLClassLoader` was created by the RMI runtime to service an invocation of one of the `RMIClassLoader.loadClass` methods, then no permissions are necessary to get the associated codebase string. If it is an arbitrary `URLClassLoader` instance, the caller must have permission to connect to all of the URLs in the codebase path, as determined by calling `openConnection().getPermission()` on each `URL` instance returned by the `getURLs` method.
- Finally, if the class loader is not an instance of `URLClassLoader`, then the value of the `java.rmi.server.codebase` property is returned, or `null` is returned if that property is not set.

The `getSecurityContext` method is deprecated because it is no longer applicable to the security model of JDK1.2; it was used internally in JDK1.1 to implement class loader-based security checks. If the indicated class loader was created by the RMI runtime to service an invocation of one of the `RMIClassLoader.loadClass` methods, the the first URL in the class loader's codebase path is returned; otherwise, null is returned.

The three `loadClass` methods all attempt to load the class with the specified name using the current thread's context class loader and, if there is a security manager set, an internal `URLClassLoader` for a particular codebase path (depending on the method):

- The `loadClass` method that only takes one parameter (the class *name*) implicitly uses the value of the `java.rmi.server.codebase` property as the codebase path to use. This version of the `loadClass` method has been deprecated because this use of the `java.rmi.server.codebase` property is discouraged; use the following, more general version instead.
- The `loadClass` method with the `String` *codebase* parameter uses it as the codebase path; the codebase string must be a space-separated list of URLs, as would be returned by the `getClassAnnotation` method.
- The `loadClass` method with the `java.net.URL` *codebase* parameter uses that single `URL` as the codebase.

For all of the `loadClass` methods, the codebase path is used in conjunction with the current thread's context class loader (determined by invoking `getContextClassLoader` on the current thread) to determine the internal class loader instance to attempt to load the class from. The RMI runtime maintains a table of internal class loader instances, keyed by the pair consisting of the parent class loader and the loader's codebase path (an ordered list of URLs). A `loadClass` method looks in the table for a `URLClassLoader` instance with the desired codebase path and the current thread's context class loader as its parent. If no such loader exists, then one is created and added to the table. Finally, the `loadClass` method is called on the chosen class loader with the specified class *name*.

If there is a security manager set (`System.getSecurityManager` does not return null), the caller of `loadClass` must have permission to connect to all of the URLs in the codebase path, or a `ClassNotFoundException` will be thrown. In order to prevent arbitrary untrusted code from being loaded into a Java VM with no security manager, if there is no security manager set, all of the `loadClass` methods will ignore the particular codebase path and only attempt to load the class with the specified *name* from the current thread's context class loader.

5.7 The LoaderHandler Interface

```
package java.rmi.server;

public interface LoaderHandler {

    Class loadClass(String name)
        throws MalformedURLException, ClassNotFoundException;
    Class loadClass(URL codebase,String name)
        throws MalformedURLException, ClassNotFoundException;
    Object getSecurityContext(ClassLoader loader);
}
```

Note – The `LoaderHandler` interface is deprecated in JDK1.2.

The `LoaderHandler` interface was only used by the JDK1.1 internal RMI implementation.

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 method `RMISocketFactory.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 JDK 1.1, 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 (for `createSocket`) and just port number (for `createServerSocket`).

In JDK 1.2, 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` and `UnicastServerRef2`, 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 an `RMIClientSocketFactory` object that is used by clients to generate socket connections to remote object at this endpoint. The endpoint representation does not contain the `RMIServerSocketFactory` 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 VM 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` and `ServerRef` 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 integer specifying the port number. This is so that RMI servers that do not use new JDK 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 `Socket` and `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 own `RMISocketFactory` 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, a `SecurityException` 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

To support custom communication with remote objects, an `RMIServerSocketFactory` instance can be specified for a remote object when it is exported, either via the appropriate `UnicastRemoteObject` constructor or `exportObject` method or the appropriate `java.rmi.activation.Activatable` constructor or `exportObject` method. If such a server socket factory is associated with a remote object when it is exported, the RMI runtime will use the remote object's server socket factory to create a `ServerSocket` (using the `RMIServerSocketFactory.createServerSocket` method) to accept connections from remote clients.

```
package java.rmi.server;
public interface RMIServerSocketFactory {

    public java.net.ServerSocket createServerSocket(int port)
        throws IOException;
}
```

5.8.3 The RMIClientSocketFactory Interface

For custom communication with remote objects, an `RMIClientSocketFactory` instance can be specified for a remote object when it is exported, either via the appropriate `UnicastRemoteObject` constructor or `exportObject` method or the appropriate `java.rmi.activation.Activatable` constructor or `exportObject` method. If such a client socket factory is associated with a remote object when it is exported, the client socket factory will be downloaded to remote virtual machines along with the remote reference for the remote object and the RMI runtime will use the `RMIClientSocketFactory.createSocket` method to make connections from the client to the remote object .

```
package java.rmi.server;
public interface RMIClientSocketFactory {
    public java.net.Socket createSocket(String host, int port)
        throws IOException;
}
```

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 a `ServerSocket` creation after a failed accept on that `ServerSocket`.

5.10 The LogStream Class

The class `LogStream` presents a mechanism for logging errors that are of possible interest to those monitoring the system. This class is used internally for server call logging.

```
package java.rmi.server;

public class LogStream extends java.io.PrintStream {
```

```
    public static LogStream log(String name);
    public static synchronized PrintStream getDefaultStream();
    public static synchronized void setDefaultStream(
        PrintStream newDefault);
    public synchronized OutputStream getOutputStream();
    public synchronized void setOutputStream(OutputStream out);
    public void write(int b);
    public void write(byte b[], int off, int len);
    public String toString();
    public static int parseLevel(String s);
    // constants for logging levels
    public static final int SILENT  = 0;
    public static final int BRIEF   = 10;
    public static final int VERBOSE = 20;
}
```

Note – The `LogStream` class is deprecated in JDK1.2

The method `log` returns the LogStream identified by the given name. If a log corresponding to name does not exist, a log using the default stream is created.

The method `getDefaultStream` returns the current default stream for new logs.

The method `setDefaultStream` sets the default stream for new logs.

The method `getOutputStream` returns current stream to which output from this log is sent.

The method `setOutputStream` sets the stream to which output from this log is sent.

The first form of the method `write` writes a byte of data to the stream. If it is not a new line, then the byte is appended to the internal buffer. If it is a new line, then the currently buffered line is sent to the log's output stream with the appropriate logging prefix. The second form of the method `write` writes a subarray of bytes.

The method `toString` returns log name as string representation.

The method `parseLevel` converts a string name of a logging level to its internal integer representation.

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. The compiler is invoked with the package qualified class name of the remote object class. The class must previously have been compiled successfully.

- The location of the imported classes may be specified either with the CLASSPATH environment variable or with the -classpath argument.
- The compiled class files are placed in the current directory unless the -d argument is specified.
- The `-keepgenerated` (or `-keep`) argument retains the generated java source files for the stubs and skeletons.
- The stub protocol version can also be specified:
  - `-v1.1` creates stubs/skeletons for the JDK 1.1 stub protocol version
  - `-vcompat` (the default in JDK 1.2) creates stubs/skeletons compatible with both JDK 1.1 and 1.2 stub protocol versions
  - `-v1.2` creates stubs for JDK 1.2 stub protocol version only (note that skeletons are not needed for the JDK 1.2 stub protocol)
- The `-show` option displays a graphical user interface for the program.
- Most `javac` command line arguments are applicable (except `-O`) and can be used with `rmic`:
  - `-g` generates debugging info
  - `-depend` recompiles out-of-date files recursively
  - `-nowarn` generates no warnings
  - `-verbose` outputs messgaes about what the compiler is doing
  - `-classpath <path>` specifies where to find input source and class files
  - `-d <directory>` specifies where to place generated class files
  - `-J<runtime flag>` passes the argument to the java interpreter

Registry Interfaces 6

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.

Topics:

- The Registry Interface
- The LocateRegistry Class
- The RegistryHandler Interface

6.1 The Registry Interface

The `java.rmi.registry.Registry` remote interface provides methods for lookup, binding, rebinding, unbinding, and listing the contents of a registry. The `java.rmi.Naming` class uses the `registry` remote interface to provide URL-based naming.

```
package java.rmi.registry;

public interface Registry extends java.rmi.Remote {
    public static final int REGISTRY_PORT = 1099;
    public java.rmi.Remote lookup(String name)
        throws java.rmi.RemoteException,
        java.rmi.NotBoundException, java.rmi.AccessException;
    public void bind(String name, java.rmi.Remote obj)
        throws java.rmi.RemoteException,
        java.rmi.AlreadyBoundException, java.rmi.AccessException;
    public void rebind(String name, java.rmi.Remote obj)
        throws java.rmi.RemoteException, java.rmi.AccessException;
    public void unbind(String name)
        throws java.rmi.RemoteException,
        java.rmi.NotBoundException, java.rmi.AccessException;
    public String[] list()
        throws java.rmi.RemoteException, java.rmi.AccessException;
}
```

The `REGISTRY_PORT` is the default port of the registry.

The `lookup` method returns the remote object bound to the specified *name*. The remote object implements a set of remote interfaces. Clients can cast the remote object to the expected remote interface. (This cast can fail in the usual ways that casts can fail in the Java language.)

The `bind` method associates the *name* with the remote object, *obj*. If the name is already bound to an object the `AlreadyBoundExcepton` is thrown.

The `rebind` method associates the *name* with the remote object, *obj*. Any previous binding of the name is discarded.

The `unbind` method removes the binding between the *name* and the remote object, *obj*. If the name is not already bound to an object the `NotBoundException` is thrown.

The `list` method returns an array of `Strings` containing a snapshot of the names bound in the registry. The return value contains a snapshot of the contents of the registry.

Clients can access the registry either by using the `LocateRegistry` and `Registry` interfaces or by using the methods of the URL-based `java.rmi.Naming` class. The registry supports `bind`, `unbind`, and `rebind` only from clients on the same host as the server; a lookup can be done from any host.

6.2 The LocateRegistry Class

The class `java.rmi.registry.LocateRegistry` is used to obtain a reference (construct a stub) to a bootstrap remote object registry on a particular host (including the local host), or to create a remote object regsitry that accepts calls on a specific port.

The registry implements a simple flat naming syntax that associates the name of a remote object (a string) with a remote object reference. The name and remote object bindings are not remembered across server restarts.

Note that a `getRegistry` call does not actually make a connection to the remote host. It simply creates a local reference to the remote registry and will succeed even if no registry is running on the remote host. Therefore, a subsequent method invocation to a remote registry return as a result of this method may fail.

```
package java.rmi.registry;
public final class LocateRegistry {
    public static Registry getRegistry()
        throws java.rmi.RemoteException;
    public static Registry getRegistry(int port)
        throws java.rmi.RemoteException;
    public static Registry getRegistry(String host)
        throws java.rmi.RemoteException;
    public static Registry getRegistry(String host, int port)
        throws java.rmi.RemoteException;
    public static Registry getRegistry(String host, int port,
                                       RMIClientSocketFactory csf)
        throws RemoteException;
    public static Registry createRegistry(int port)
        throws java.rmi.RemoteException;
    public static Registry createRegistry(int port,
                                      RMIClientSocketFactory csf,
                                      RMIServerSocketFactory ssf)
        throws RemoteException;
}
```

The first four `getRegistry` methods return a reference to a registry on the current host, current host at a specified *port*, a specified *host,* or at a particular *port* on a specified *host*. What is returned is the remote stub for the registry with the specified host and port information.

The fifth `getRegistry` method (that takes an `RMIClientSocketFactory` as one of its arguments), returns a locally created remote stub to the remote object `Registry` on the specified *host* and *port*. Communication with the remote registry whose stub is constructed with this method will use the supplied `RMIClientSocketFactory`, *csf*, to create `Socket` connections to the registry on the remote host and port.

Note – A registry returned from the `getRegistry` methods is a specially constructed stub that contains a well-known object identifier. Passing a registry stub from one VM to another is not supported (it may or may not work depending on the implementation). Use the `LocateRegistry.getRegistry` methods to obtain the appropriate registry for a host.

The `createRegistry` methods creates and exports a registry on the local host on the specified *port*.

The second `createRegistry` method allows more flexiblity in communicating with the registry. This call creates and exports a `Registry` on the local host that uses custom socket factories for communication with that registry. The registry that is created listens for incoming requests on the given *port* using a `ServerSocket` created from the supplied `RMIServerSocketFactory`. A client that receives a reference to this registry will use a `Socket` created from the supplied `RMIClientSocketFactory`.

Note – Starting a registry with the `createRegistry` method does not keep the server process alive.

6.3 The RegistryHandler Interface

Note – The RegistryHandler interface is deprecated in JDK1.2. In JDK1.1, it was only used internally by the RMI implementation and was not for application use.

```
package java.rmi.registry;

public interface RegistryHandler {
    Registry registryStub(String host, int port)
        throws java.rmi.RemoteException,
        java.rmi.UnknownHostException;
```

```
    Registry registryImpl(int port)
        throws java.rmi.RemoteException;
}
```

The method `registryStub` returns a stub for contacting a remote registry on the specified host and port.

The method `registryImpl` constructs and exports a registry on the specified port. The port must be nonzero.

Remote Object Activation 7

Topics:

- Overview
- Activation Protocol
- Implementation Model for an “Activatable” Remote Object
- Activation Interfaces

7.1 Overview

Distributed object systems are designed to support long-lived persistent objects. Given that these systems will be made up of many thousands (perhaps millions) of such objects, it would be unreasonable for object implementations to become active and remain active, taking up valuable system resources, for indefinite periods of time. In addition, clients need the ability to store persistent references to objects so that communication among objects can be re-established after a system crash, since typically a reference to a distributed object is valid only while the object is active.

Object activation is a mechanism for providing persistent references to objects and managing the execution of object implementations. In RMI, activation allows objects to begin execution on an as-needed basis. When an “activatable” remote object is accessed (via a method invocation) if that remote object is not currently executing, the system initiates the object's execution inside an appropriate Java VM.

7.1.1 Terminology

An *active* object is a remote object that is instantiated and exported in a Java VM on some system. A *passive* object is one that is not yet instantiated (or exported) in a VM, but which can be brought into an active state. Transforming a passive object into an active object is a process known as *activation*. Activation requires that an object be associated with a VM, which may entail loading the class for that object into a VM and the object restoring its persistent state (if any).

In the RMI system, we use *lazy activation*. Lazy activation defers activating an object until a client's first use (i.e., the first method invocation).

7.1.2 Lazy Activation

Lazy activation of remote objects is implemented using a *faulting remote reference* (sometimes referred to as a fault block). A faulting remote reference to a remote object "faults in" the active object's reference upon the first method invocation to the object. Each faulting reference maintains both a persistent handle (an activation identifier) and a transient remote reference to the target remote object. The remote object's activation identifier contains enough information to engage a third party in activating the object. The transient reference is the actual "live" reference to the active remote object that can be used to contact the executing object.

In a faulting reference, if the live reference to a remote object is null, the target object is not known to be active. Upon method invocation, the faulting reference (for that object) engages in the activation protocol to obtain a "live" reference, which is a remote reference (such as a unicast remote reference) for the newly-activated object. Once the faulting reference obtains the live reference, the faulting reference forwards method invocations to the underlying remote reference which, in turn, forwards the method invocation to the remote object.

In more concrete terms, a remote object's stub contains a "faulting" remote reference type that contains both:

- an activation identifier for a remote object, and
- a "live" reference (possibly null) containing the "active" remote reference type of the remote object (for example, a remote reference type with unicast semantics).

Note – The RMI system preserves "at most once" semantics for remote calls. In other words, a call to an *activatable* or *unicast* remote object is sent at most once. Thus, if a call to a remote object fails (indicated by a `RemoteException` being thrown), the client can be guaranteed that the remote method executed no more than once (and perhaps not at all).

7.2 Activation Protocol

During a remote method invocation, if the "live" reference for a target object is unknown, the faulting reference engages in the activation protocol. The activation protocol involves several entities: the faulting reference, the *activator*, an *activation group* in a Java VM, and the remote object being activated.

The activator (usually one per host) is the entity which supervises activation by being both:

- a database of information that maps activation identifiers to the information necessary to activate an object (the object's class, the location--a URL path--from where the class can be loaded, specific data the object may need to bootstrap, etc.), and
- a manager of Java Virtual Machines, that starts up VMs (when necessary) and forwards requests for object activation (along with the necessary information) to the correct activation group inside a remote VM.

Note that the activator keeps the current mapping of activation identifiers to active objects as a cache, so that the group does not need to be consulted on each activation request.

An activation group (one per Java VM) is the entity which receives a request to activate an object in the Java VM and returns the activated object back to the activator.

The activation protocols is as follows. A faulting reference uses an activation identifier and calls the activator (an internal RMI interface) to activate the object associated with the identifier. The activator looks up the object's *activation descriptor* (registered previously). The object's descriptor contains:

- the object's group identifier (specifies the VM in which it is activated),
- the object's class name,
- a URL path from where to load the object's class code,

- object-specific initialization data in marshalled form (initialization data might be the name of a file containing the object's persistent state, for example).

If the activation group in which this object should reside exists, the activator forwards the activation request to that group. If the activation group does not exist, the activator initiates a VM executing an activation group and then forwards the activation request to that group.

The activation group loads the class for the object and instantiates the object using a special constructor that takes several arguments, including the activation descriptor registered previously.

When the object is finished activating, the activation group passes back a *marshalled object* reference to the activator that then records the activation identifier and active reference pairing and returns the active (live) reference to the faulting reference. The faulting reference (inside the stub) then forwards method invocations via the live reference directly to the remote object.

Note – In the JDK, RMI provides an implementation of the activation system interfaces. In order to use activation, you must first run the activation system daemon **`rmid`**.

7.3 Implementation Model for an "Activatable" Remote Object

In order to make a remote object that can be accessed via an activation identifier over time, a developer needs to:

- register an activation descriptor for the remote object, and
- include a special constructor in the object's class that the RMI system calls when it activates the activatable object.

An activation descriptor (`ActivationDesc`) can be registered in one of several ways:

- via a call to the static `register` method of the class `Activatable`, or
- by creating an "activatable" object via the first or second constructor of the `Activatable` class, or
- by exporting an "activatable" object explicitly via `Activatable`'s first or second `exportObject` method that takes an `ActivationDesc`, the `Remote` object implementation and a port number as arguments.

For a specific object, only one of the above methods should be used to register the object for activation. See the section below on “Constructing an Activatable Remote Object” for examples on how to implement activatable objects.

7.3.1 The ActivationDesc Class

An `ActivationDesc` contains the information necessary to activate an object. It contains the object’s activation group identifier, the class name for the object, a codebase path (or URLs) from where the object’s code can be loaded, and a `MarshalledObject` that may contain object-specific initialization data used during each activation.

A descriptor registered with the activation system is consulted (during the activation process) to obtain information in order to re-create or activate an object. The `MarshalledObject` in the object’s descriptor is passed as the second argument to the remote object’s constructor for the object to use during activation.

```
package java.rmi.activation;
public final class ActivationDesc implements java.io.Serializable
{

    public ActivationDesc(String className,
                          String codebase,
                          java.rmi.MarshalledObject data)
        throws ActivationException;

    public ActivationDesc(String className,
                          String codebase,
                          java.rmi.MarshalledObject data,
                          boolean restart)
        throws ActivationException;

    public ActivationDesc(ActivationGroupID groupID,
                          String className,
                          String codebase,
                          java.rmi.MarshalledObject data,
                          boolean restart);

    public ActivationDesc(ActivationGroupID groupID,
                          String className,
                          String codebase,
```

```
                            java.rmi.MarshalledObject data);

    public ActivationGroupID getGroupID();
    public String getClassName();
    public String getLocation();
    public java.rmi.MarshalledObject getData()
    public boolean getRestartMode();
}
```

The first constructor for `ActivationDesc` constructs an object descriptor for an object whose class is *className*, that can be loaded from *codebase* path, and whose initialization information, in marshalled form, is *data*. If this form of the constructor is used, the object's group identifier defaults to the current identifier for `ActivationGroup` for this VM. All objects with the same `ActivationGroupID` are activated in the same VM. If the current group is inactive or a default group cannot be created an `ActivationException` is thrown. If the *groupID* is null, an `IllegalArgumentException` is thrown.

Note – As a side-effect of creating an `ActivationDesc`, if an `ActivationGroup` for this VM is not currently active, a default one is created. The default activation group uses the `java.rmi.RMISecurityManager` as a security manager and upon reactivation will set the properties in the activated group's VM to be the current set of properties in the VM. If your application needs to use a different security manager, it must set the group for the VM before creating a default `ActivationDesc`. See the method `ActivationGroup.createGroup` for details on how to create an `ActivationGroup` for the VM.

The second constructor for `ActivationDesc` constructs an object descriptor in the same manner as the first constructor except an additional parameter, *restart*, must be supplied. If the object requires *restart service*, meaning that the object will be restarted automatically when the activator is restarted (as opposed to being activated lazily upon demand), *restart* should be true. If *restart* is false, the object is simply activated upon demand (via a remote method call).

The third constructor for `ActivationDesc` constructs an object descriptor for an object whose group identifier is *groupID*, whose class name is *className* that can be loaded from the *codebase* path, and whose initialization information is *data*. All objects with the same *groupID* are activated in the same Java VM.

The fourth constructor for `ActivationDesc` constructs an object descriptor in the same manner as the third constructor, but allows a restart mode to be specified. An object requires restart service (as defined above), *restart* should be true.

The `getGroupID` method returns the group identifier for the object specified by the descriptor. A group provides a way to aggregate objects into a single Java virtual machine.

The `getClassName` method returns the class name for the object specified by the activation descriptor.

The `getLocation` method returns the codebase path from where the object's class can be downloaded.

The `getData` method returns a "marshalled object" containing initialization (activation) data for the object specified by the descriptor.

The `getRestartMode` method returns true if the restart mode is enabled for this object, otherwise it returns false.

7.3.2 The ActivationID Class

The activation protocol makes use of activation identifiers to denote remote objects that can be activated over time. An activation identifier (an instance of the class `ActivationID`) contains several pieces of information needed for activating an object:

- a remote reference to the object's activator, and
- a unique identifier for the object.

An activation identifier for an object can be obtained by registering an object with the activation system. Registration is accomplished in a few ways (also noted above):

- via the `Activatable.register` method, or
- via the first or second `Activatable` constructor (that takes three arguments and both registers and exports the object), or
- via the first or second `Activatable.exportObject` method that takes the activation descriptor, object implementation, and port as arguments; this method both registers and exports the object.

```
package java.rmi.activation;
public class ActivationID implements java.io.Serializable
{
    public ActivationID(Activator activator);

    public Remote activate(boolean force)
        throws ActivationException, UnknownObjectException,
               java.rmi.RemoteException;

    public boolean equals(Object obj);

    public int hashCode();
}
```

The constructor for `ActivationID` takes a single argument, *activator*, that specifies a remote reference to the activator responsible for activating the object associated with this activation identifier. An instance of `ActivationID` is globally unique.

The `activate` method activates the object associated with the activation identifier. If the *force* parameter is true, the activator considers any cached reference for the remote object as stale, thus forcing the activator to contact the group when activating the object. If *force* is false, then returning the cached value is acceptable. If activation fails, `ActivationException` is thrown. If the object identifier is not known to the activator, then the method throws `UnknownObjectException`. If the remote call to the activator fails, then `RemoteException` is thrown.

The `equals` method implements content equality. It returns true if all fields are equivalent (either identical or equivalent according to each field's `Object.equals` semantics). If *p1* and *p2* are instances of the class `ActivationID`, the `hashCode` method will return the same value if `p1.equals(p2)` returns true.

7.3.3 The Activatable Class

The `Activatable` class provides support for remote objects that require persistent access over time and that can be activated by the system. The class `Activatable` is the main API that developers need to use to implement and manage activatable objects. Note that you must first run the activation system daemon, **`rmid`**, before objects can be registered and/or activated.

```
package java.rmi.activation;
public abstract class Activatable
    extends java.rmi.server.RemoteServer
{
    protected Activatable(String codebase,
                          java.rmi.MarshalledObject data,
                          boolean restart,
                          int port)
        throws ActivationException, java.rmi.RemoteException;

    protected Activatable(String codebase,
                          java.rmi.MarshalledObject data,
                          boolean restart,
                          int port,
                          RMIClientSocketFactory csf,
                          RMIServerSocketFactory ssf)
        throws ActivationException, java.rmi.RemoteException;

    protected Activatable(ActivationID id, int port)
        throws java.rmi.RemoteException;

    protected Activatable(ActivationID id, int port,
                          RMIClientSocketFactory csf,
                          RMIServerSocketFactory ssf)
        throws java.rmi.RemoteException;

    protected ActivationID getID();

    public static Remote register(ActivationDesc desc)
        throws UnknownGroupException, ActivationException,
               java.rmi.RemoteException;

    public static boolean inactive(ActivationID id)
        throws UnknownObjectException, ActivationException,
               java.rmi.RemoteException;

    public static void unregister(ActivationID id)
        throws UnknownObjectException, ActivationException,
               java.rmi.RemoteException;

    public static ActivationID exportObject(Remote obj,
                                            String codebase,
                                            MarshalledObject data,
                                            boolean restart,
                                            int port)
        throws ActivationException, java.rmi.RemoteException;
```

```
    public static ActivationID exportObject(Remote obj,
                                              String codebase,
                                              MarshalledObject data,
                                              boolean restart,
                                              int port,
                                           RMIClientSocketFactory csf,
                                           RMIServerSocketFactory ssf)
        throws ActivationException, java.rmi.RemoteException;

    public static Remote exportObject(Remote obj,
                                       ActivationID id,
                                       int port)
        throws java.rmi.RemoteException;

    public static Remote exportObject(Remote obj,
                                       ActivationID id,
                                       int port,
                                       RMIClientSocketFactory csf,
                                       RMIServerSocketFactory ssf)
        throws java.rmi.RemoteException;

    public static boolean unexportObject(Remote obj, boolean force)
        throws java.rmi.NoSuchObjectException;
}
```

An implementation for an activatable remote object may or may not extend the class `Activatable`. A remote object implementation that *does* extend the `Activatable` class inherits the appropriate definitions of the `hashCode` and `equals` methods from the superclass `java.rmi.server.RemoteObject`. So, two remote object references that refer to the same `Activatable` remote object will be equivalent (the `equals` method will return `true`). Also, an instance of the class `Activatable` will be "equals" to the appropriate stub object for the instance (i.e., the `Object.equals` method will return `true` if called with the matching stub object for the implementation as an argument, and vice versa).

Activatable Class Methods

The first constructor for the `Activatable` class is used to register and export the object on a specified *port* (an anonymous port is chosen if *port* is zero). The object's URL path for downloading its class code is *codbase*, and its initialization data is *data*. If *restart* is true, the object will be restarted automatically when the activator is restarted and if the group crashes. If *restart* is false, the object will be activated on demand (via a remote method call to the object).

A concrete subclass of the `Activatable` class must call this constructor to register and export the object during *initial* construction. As a side-effect of activatable object construction, the remote object is both "registered" with the activation system and "exported" (on an anonymous port, if *port* is zero) to the RMI runtime so that it is available to accept incoming calls from clients.

The constructor throws `ActivationException` if registering the object with the activation system fails. `RemoteException` is thrown if exporting the object to the RMI runtime fails.

The second constructor is the same as the first `Activatable` constructor but allows the specification of the client and server socket factories used to communicate with this activatable object. See the section in about "RMI Socket Factories" for details.

The third constructor is used to activate and export the object (with the `ActivationID`, *id*) on a specified *port*. A concrete subclass of the `Activatable` class must call this constructor when the object itself is *activated* via its special "activation" constructor whose parameters must be:

- the object's activation identifier (`ActivationID`), and
- the object's initialization/bootstrap data (a `MarshalledObject`).

As a side-effect of construction, the remote object is "exported" to the RMI runtime (on the specified *port*) and is available to accept incoming calls from clients. The constructor throws `RemoteException` if exporting the object to the RMI runtime fails.

The fourth constructor is the same as the third constructor, but allows the specification of the client and server socket factories used to communicate with this activatable object.

The `getID` method returns the object's activation identifier. The method is protected so that only subclasses can obtain and object's identifier. The object's identifier is used to report the object as inactive or to unregister the object's activation descriptor.

The `register` method registers, with the activation system, an object descriptor, *desc*, for an activatable remote object so that it can be activated on demand. This method is used to register an activatable object without having to first create the object. This method returns the `Remote` stub for the activatable object so that it can be saved and called at a later time thus forcing the object to be created/activated for the first time. The method throws `UnknownGroupException` if the group identifier in *desc* is not registered with the activation system. `ActivationException` is thrown if the activation system is not running. Finally, `RemoteException` is thrown if the remote call to the activation system fails.

The `inactive` method is used to inform the system that the object with the corresponding activation *id* is currently inactive. If the object is currently known to be active, the object is unexported from the RMI runtime (only if there are no pending or executing calls) so the that it can no longer receive incoming calls. This call also informs this VM's `ActivationGroup` that the object is inactive; the group, in turn, informs its `ActivationMonitor`. If the call completes successfully, subsequent activate requests to the activator will cause the object to reactivate. The `inactive` method returns *true* if the object was successfully unexported (meaning that it had no pending or executing calls at the time) and returns *false* if the object could not be unexported due to pending or in-progress calls. The method throws `UnknownObjectException` if the object is not known (it may already be inactive); an `ActivationException` is thrown if the group is not active; a `RemoteException` is thrown if the call informing the monitor fails. The operation may still succeed if the object is considered active but has already unexported itself.

The `unregister` method revokes previous registration for the activation descriptor associated with *id*. An object can no longer be activated via that *id*. If the object *id* is unknown to the activation system, a `UnknownObjectException` is thrown. If the activation system is not running an `ActivationException` is thrown. If the remote call to the activation system fails, then a `RemoteException` is thrown.

The first `exportObject` method may be invoked explicitly by an "activatable" object, that does not extend the `Activatable` class, in order to both a) register the object's activation descriptor, *desc*, constructed from the supplied *codebase* and *data*, with the activation system (so the object can be activated), and b) export the remote object, *obj*, on a specific *port* (if the *port* is zero, then an anonymous port is chosen). Once the object is exported, it can receive incoming RMI calls.

This `exportObject` method returns the activation identifier obtained from registering the descriptor, *desc*, with the activation system. If the activation group is not active in the VM, then `ActivationException` is thrown. If the object registration or export fails, then `RemoteException` is thrown.

This method does not need to be called if *obj* extends `Activatable`, since the first `Activatable` constructor calls this method.

The second `exportObject` method is the same as the first except it allows the specification of client and server socket factories used to communicate with the activatable object.

The third `exportObject` method exports an "activatable" remote object (not necessarily of type `Activatable`) with the identifier, *id*, to the RMI runtime to make the object, *obj*, available to receive incoming calls. The object is exported on an anonymous port, if *port* is zero.

During activation, this `exportObject` method should be invoked explicitly by an "activatable" object, that does not extend the `Activatable` class. There is no need for objects that do extend the `Activatable` class to invoke this method directly; this method is called by the third constructor above (which a subclass should invoke from its special activation constructor).

This `exportObject` method returns the `Remote` stub for the activatable object. If the object export fails, then the method throws `RemoteException`.

The fourth exportObject method is the same as the third but allows the specification of the client and server socket factories used to communicate with this activatable object.

The `unexportObject` method makes the remote object, *obj*, unavailable for incoming calls. If the force parameter is true, the object is forcibly unexported eve 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. Removing the object from RMI use 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.

Constructing an Activatable Remote Object

In order for an object to be activated, the "activatable" object implementation class (whether or not it extends the `Activatable` class) must define a special public constructor that takes two arguments, its activation identifier of type `ActivationID`, and its activation data, a `java.rmi.MarshalledObject`, supplied in the activation descriptor used during registration. When an activation group activates a remote object inside its VM, it constructs the object via this special constructor (described in more detail below). The remote object implementation may use the activation data to initialize itself in a suitable manner. The remote object may also wish to retain its activation identifier, so that it can inform the activation group when it becomes inactive (via a call to the `Activatable.inactive` method).

The first and second constructor forms for `Activatable` is used to both register and export an activatable object on a specified *port*. This constructor should be used when initially constructing the object; the third form of the constructor is used when re-activating the object.

A concrete subclass of `Activatable` must call the first or second constructor form to register and export the object during initial construction. This constructor first creates an activation descriptor (`ActivationDesc`) with the object's class name, the object's supplied *codebase* and *data*, and whose activation group is the default group for the VM. Next, the constructor registers this descriptor with the default `ActivationSystem`. Finally, the constructor exports the activatable object to the RMI runtime on the specific *port* (if *port* is zero, then an anonymous port is chosen) and reports the object as an `activeObject` to the local `ActivationGroup`. If an error occurs during registration or export, the constructor throws `RemoteException`. Note that the constructor also initializes its `ActivationID` (obtained via registration), so that subsequent calls to the protected method `getID` will return the object's activation identifier.

The third constructor form for `Activatable` is used to export the object on a specified port. A concrete subclass of `Activatable` must call the third constructor form when it is activated via the object's own "activation" constructor which takes two arguments:

- the object's `ActivationID`
- the object's initialization data, a `MarshalledObject`

This constructor only exports the activatable object to the RMI runtime on the specific *port* (if *port* is 0, then an anonymous port is chosen), it does not inform the `ActivationGroup` that the object is active, since it is the `ActivationGroup` that is activating the object and knows it to be active already.

The following is an example of a remote object interface, `Server`, and an implementation, `ServerImpl`, that *extends* the `Activatable` class:

```
package examples;

public interface Server extends java.rmi.Remote {
    public void doImportantStuff()
        throws java.rmi.RemoteException;
}


public class ServerImpl extends Activatable implements Server
{
    // Constructor for initial construction, registration and export
    public ServerImpl(String codebase, MarshalledObject data)
        throws ActivationException, java.rmi.RemoteException
    {
        // register object with activation system, then
        // export on anonymous port
        super(codebase, data, false, 0);
    }

    // Constructor for activation and export; this constructor
    // is called by the ActivationInstantiator.newInstance
    // method during activation in order to construct the object.
    public ServerImpl(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException
    {
        // call the superclass's constructor in order to
        // export the object to the RMI runtime.
        super(id, 0);
        // initialize object (using data, for example)
    }

    public void doImportantStuff() { ... }
}
```

An object is responsible for exporting itself. The constructors for `Activatable` take care of *exporting* the object to the RMI runtime with the live reference type of a `UnicastRemoteObject`, so the object implementation extending `Activatable` does not need to worry about the detail of exporting the object explicitly (other than invoking the appropriate superclasses constructor). If an object implementation does not extend the class `Activatable`, the object must export the object explicitly via a call to one of the `Activatable.exportObject` static methods,.

In the following example, `ServerImpl` does *not extend* `Activatable`, but rather another class, so `ServerImpl` is responsible for exporting itself during initial construction and activation. The following class definition shows `ServerImpl`'s initialization constructor and its special "activation" constructor and the appropriate call to export the object within each constructor:

```
package examples;
public class ServerImpl extends SomeClass implements Server
{
    // constructor for initial creation
    public ServerImpl(String codebase, MarshalledObject data)
        throws ActivationException, java.rmi.RemoteException
    {
        // register and export the object
        Activatable.exportObject(this, codebase, data, false, 0);
    }

    // constructor for activation
    public ServerImpl(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException
    {
        // export the object
        Activatable.exportObject(this, id, 0);
    }

    public void doImportantStuff() { ... }
}
```

Registering an Activation Descriptor Without Creating the Object

To register an activatable remote object with the activation system without first creating the object, the programmer can simply register an activation descriptor (an instance of the class `ActivationDesc`) for the object. An activation descriptor contains all the necessary information so that the activation system can activate the object when needed. An activation descriptor for an instance of the class `examples.ServerImpl` can be registered in the following manner (exception handling elided):

```
Server server;
ActivationDesc desc;
String codebase = "http://zaphod/codebase/";

MarshalledObject data = new MarshalledObject("some data");

desc = new ActivationDesc("examples.ServerImpl", codebase, data);
server = (Server)Activatable.register(desc);
```

The `register` call returns a `Remote` stub that is the stub for the `examples.ServerImpl` object and implements the same set of remote interfaces that `examples.ServerImpl` implements (i.e, the stub implements the remote interface `Server`). This stub object (above, cast and assigned to *server*) can be passed as a parameter in any method call expecting an object that implements the `examples.Server` remote interface.

7.4 Activation Interfaces

In the RMI activation protocol, there are two guarantees that the activator must make for the system to function properly:

- like all system daemons, the activator should remain running while the machine is up, and
- the activator must not reactivate remote objects that are already active.

The activator maintains a database of appropriate information for the groups and objects that it participates in activating.

7.4.1 The Activator Interface

The activator is one of the entities that participates during the activation process. As described earlier, a faulting reference (inside a stub) calls the activator's `activate` method to obtain a "live" reference to an activatable remote object. Upon receiving a request for activation, the activator looks up the activation descriptor for the activation identifier, *id*, determines the group in which the object should be activated and invokes the `newInstance` method on the activation group's instantiator (the remote interface `ActivationGroup` is described below). The activator initiates the execution of activation groups as necessary. For example, if an activation group for a specific group descriptor is not already executing, the activator will spawn a child VM for the activation group to establish the group in the new VM.

The activator is responsible for monitoring and detecting when activation groups fail so that it can remove stale remote references from its internal tables.

```
package java.rmi.activation;
public interface Activator extends java.rmi.Remote
{
    java.rmi.MarshalledObject activate(ActivationID id,
                                       boolean force)
        throws UnknownObjectException, ActivationException,
               java.rmi.RemoteException;
}
```

The `activate` method activates the object associated with the activation identifier, *id*. If the activator knows the object to be active already and the *force* parameter is `false`, the stub with a "live" reference is returned immediately to the caller; otherwise, if the activator does not know that corresponding the remote object is active or the *force* parameter is `true`, the activator uses the activation descriptor information (previously registered to obtain the *id*) to determine the group (VM) in which the object should be activated. If an `ActivationInstantiator` corresponding to the object's group already exists, the activator invokes the activation instantiator's `newInstance` method passing it the *id* and the object's activation descriptor.

If the activation instantiator (group) for the object's group descriptor does not yet exist, the activator starts a new incarnation of an `ActivationInstantiator` executing (by spawning a child process, for example). When the activator re-creates an ActivationInstantiator for a group, it must increment the group's incarnation number. Note that the incarnation number is zero-based. The activation system uses incarnation numbers to detect late `ActivationSystem.activeGroup` and `ActivationMonitor.inactiveGroup` calls. The activation system discards calls with an earlier incarnation number than the current number for the group.

Note – The activator must communicate both the activation group's identifier, descriptor and incarnation number when it starts up a new activation group. The activator spawns an activation group in a separate VM (as a separate or child process, for example), and therefore must pass information specifying the information necessary to create the group via the

ActivationGroup.createGroup method. How the activator sends this information to the spawned process is unspecified, however, this information could be sent in the form of marshalled objects to the child process's *standard input*.

When the activator receives the activation group's call back (via the ActivationSystem.activeGroup method) specifying the activation group's reference and incarnation number, the activator can then invoke that activation instantiator's newInstance method to forward each pending activation request to the activation instantiator and return the result (a *marshalled* remote object reference, a stub) to each caller.

Note that the activator receives a MarshalledObject instead of a Remote object so that the activator does not need to load the code for that object, or participate in distributed garbage collection for that object. If the activator kept a strong reference to the remote object, the activator would then prevent the object from being garbage collected under the normal distributed garbage collection mechanism.

The activate method throws ActivationException if activation fails. Activation may fail for a variety of reasons: the class could not be found, the activation group could not be contacted, etc. The activate method throws UnknownObjectException if no activation descriptor for the activation identifier, *id*, has been previously registered with this activator. RemoteException is thrown if the remote call to the activator fails.

7.4.2 The ActivationSystem Interface

The ActivationSystem provides a means for registering groups and *activatable* objects to be activated within those groups. The ActivationSystem works closely with both the Activator, which activates objects registered via the ActivationSystem, and the ActivationMonitor, which obtains information about active and inactive objects and inactive groups.

```
package java.rmi.activation;
public interface ActivationSystem extends java.rmi.Remote
{
    public static final int SYSTEM_PORT = 1098;

    ActivationGroupID registerGroup(ActivationGroupDesc desc)
        throws ActivationException, java.rmi.RemoteException;
```

```
    ActivationMonitor activeGroup(ActivationGroupID id,
                                  ActivationInstantiator group,
                                  long incarnation)
        throws UnknownGroupException, ActivationException,
               java.rmi.RemoteException;

    void unregisterGroup(ActivationGroupID id)
        throws ActivationException, UnknownGroupException,
               java.rmi.RemoteException;

    ActivationID registerObject(ActivationDesc desc)
        throws ActivationException, UnknownGroupException,
               java.rmi.RemoteException;

    void unregisterObject(ActivationID id)
        throws ActivationException, UnknownObjectException,
               java.rmi.RemoteException;

    void shutdown() throws java.rmi.RemoteException;
}
```

Note – As a security measure, all of the above methods (`registerGroup`, `activeGroup`, `unregisterGroup`, `registerObject`, `unregisterObject`, and `shutdown`) will throw `java.rmi.AccessException`, a subclass of `java.rmi.RemoteException` if called from a client that does not reside on the same host as the activation system.

The `registerObject` method is used to register an activation descriptor, *desc*, and obtain an activation identifier for an activatable remote object. The `ActivationSystem` creates an `ActivationID` (an activation identifier) for the object specified by the descriptor, *desc*, and records, in stable storage, the activation descriptor and its associated identifier for later use. When the `Activator` receives an `activate` request for a specific identifier, it looks up the activation descriptor (registered previously) for the specified identifier and uses that information to activate the object. If the group referred to in *desc* is not registered with this system, then the method throws `UnknownGroupException`. If registration fails (e.g., database update failure, etc), then the method throws ActivationException. If the remote call fails, then `RemoteException` is thrown.

The `unregisterObject` method removes the activation identifier, *id*, and associated descriptor previously registered with the `ActivationSystem`. After the call completes, the object can no longer be activated via the object's activation *id*. If the object *id* is unknown (not registered) the method throws `UnknownObjectException`. If the unregister operation fails (e.g., database update failure, etc), then the method throws ActivationException. If the remote call fails, then `RemoteException` is thrown.

The `registerGroup` method registers the activation group specified by the group descriptor, *desc*, with the activation system and returns the `ActivationGroupID` assigned to that group. An activation group must be registered with the `ActivationSystem` before objects can be registered within that group. If group registration fails, the method throws `ActivationException`. If the remote call fails then `RemoteException` is thrown.

The `activeGroup` method is a call back from the `ActivationGroup` (with the identifier, *id)*, to inform the activation system that *group* is now active and is the `ActivationInstantiator` for that VM. This call is made internally by the `ActivationGroup.createGroup` method to obtain an `ActivationMonitor` that the group uses to update the system regarding objects' and the group's status (i.e., that the group or objects within that group have become inactive). If the group is not registered, then the method throws `UnknownGroupException`. If the group is already active, then `ActivationException` is thrown. If the remote call to the activation system fails, then `RemoteException` is thrown.

The `unregisterGroup` method removes the activation group with identifier, *id*, from the activation system. An activation group makes this call back to inform the activator that the group should be destroyed. If this call completes successfully, objects can no longer be registered or activated within the group. All information of the group and its associated objects is removed from the system. The method throws `UnknownGroupException` if the group is not registered. If the remote call fails, then `RemoteException` is thrown. If the unregister fails, ActivationException is thrown (e.g., database update failure, etc.).

The `shutdown` method gracefully terminates (asynchronously) the activation system and all related activation processes (activator, monitors and groups). All groups spawned by the activation daemon will be destroyed and the activation daemon will exit. In order to shut down the activation system daemon, rmid, execute the command:

```
rmid -stop [-port num]
```

This command will shut down the activation daemon on the specified port (if no port is specified, the daemon on the default port will be shut down).

7.4.3 The ActivationMonitor Class

An `ActivationMonitor` is specific to an `ActivationGroup` and is obtained when a group is reported via a call to `ActivationSystem.activeGroup` (this is done internally by the `ActivationGroup.createGroup` method). An activation group is responsible for informing its `ActivationMonitor` when either: its objects become active, inactive or the group as a whole becomes inactive.

```
package java.rmi.activation;
public interface ActivationMonitor
    extends java.rmi.Remote
{

    public abstract void inactiveObject(ActivationID id)
        throws UnknownObjectException, RemoteException;

    public void activeObject(ActivationID id,
                              java.rmi.MarshalledObject mobj)
        throws UnknownObjectException, java.rmi.RemoteException;

    public void inactiveGroup(ActivationGroupID id,
                               long incarnation)
        throws UnknownGroupException, java.rmi.RemoteException;
}
```

An activation group calls its monitor's `inactiveObject` method when an object in its group becomes inactive (deactivates). An activation group discovers that an object (that it participated in activating) in its VM is no longer active via a call to the activation group's `inactiveObject` method.

The `inactiveObject` call informs the `ActivationMonitor` that the remote object reference it holds for the object with the activation identifier, *id*, is no longer valid. The monitor considers the reference associated with *id* as a stale reference. Since the reference is considered stale, a subsequent `activate` call for the same activation identifier results in re-activating the remote object. If the object is not known to the `ActivationMonitor`, the method throws `UnknownObjectException`. If the remote call fails, then `RemoteException` is thrown.

The activeObject call informs the ActivationMonitor that the object associated with *id* is now active. The parameter *obj* is the marshalled representation of the object's stub. An ActivationGroup must inform its monitor if an object in its group becomes active by other means than being activated directly by the system (i.e., the object is registered and "activated" itself). If the object id is not previously registered, then the method throws UnknownObjectException. If the remote call fails, then RemoteException is thrown.

The inactiveGroup call informs the monitor that the group specified by *id* and *incarnation* is now inactive. The group will be re-created with a greater incarnation number upon a subsequent request to activate an object within the group. A group becomes inactive when all objects in the group report that they are inactive. If either the group *id* is not registered or the incarnation number is smaller than the current incarnation for the group, then the method throws UnknownGroupException. If the remote call fails, then RemoteException is thrown.

7.4.4 The ActivationInstantiator Class

The ActivationInstantiator is responsible for creating instances of activatable objects. A concrete subclass of ActivationGroup implements the newInstance method to handle creating objects within the group.

```
package java.rmi.activation;
public interface ActivationInstantiator
    extends java.rmi.Remote
{
    public MarshalledObject newInstance(ActivationID id,
                                        ActivationDesc desc)
        throws ActivationException, java.rmi.RemoteException;

}
```

The activator calls an instantiator's newInstance method in order to re-create in that group an object with the activation identifier, *id*, and descriptor, *desc*. The instantiator is responsible for:

- determining the class for the object using the descriptor's getClassName method,

- loading the class from the codebase path obtained from the descriptor (using the `getLocation` method),
- creating an instance of the class by invoking the special "activation" constructor of the object's class that takes two arguments: the object's `ActivationID`, and the `MarshalledObject` containing object-specific initialization data, and
- returning a `MarshalledObject` containing the remote object it created.

An instantiator is also responsible for reporting when objects it creates or activates are no longer active, so that it can make the appropriate `inactiveObject` call to its `ActivationMonitor` (see the `ActivationGroup` class for more details).

If object activation fails, then the `newInstance` method throws `ActivationException`. If the remote call fails, then the method throws `RemoteException`.

7.4.5 The ActivationGroupDesc Class

An activation group descriptor (`ActivationGroupDesc`) contains the information necessary to create or re-create an activation group in which to activate objects in the same Java VM.

Such a descriptor contains:

- the group's class name,
- the group's codebase path (the location of the group's class), and
- a "marshalled" object that can contain object-specific initialization data.

The group's class must be a concrete subclass of `ActivationGroup`. A subclass of `ActivationGroup` is created or re-created via the `ActivationGroup.createGroup` static method that invokes a special constructor that takes two arguments:

- the group's `ActivationGroupID`, and
- the group's initialization data (in a `java.rmi.MarshalledObject`)

```
package java.rmi.activation;
public final class ActivationGroupDesc
    implements java.io.Serializable
{
```

```
    public ActivationGroupDesc(java.util.Properties props,
                               CommandEnvironment env);;

    public ActivationGroupDesc(String className,
                               String codebase,
                               java.rmi.MarshalledObject data,
                               java.util.Properties props,
                               CommandEnvironment env);

    public String getClassName();

    public String getLocation();

    public java.rmi.MarshalledObject getData();

    public CommandEnvironment getCommandEnvironment();

    public java.util.Properties getPropertiesOverrides();

}
```

The first constructor creates a group descriptor that uses system default for group implementation and code location. Properties specify Java environment overrides (which will override system properties in the group implementation's VM). The command environment can control the exact command/options used in starting the child VM, or can be *null* to accept `rmid`'s default.

The second constructor is the same as the first, but allows the specification of `Properties` and `CommandEnvironment`.

The `getClassName` method returns the group's class name.

The `getLocation` method returns the codebase path from where the group's class can be loaded.

The `getData` method returns the group's initialization data in marshalled form.

The `getCommandEnvironment` method returns the command environment (possibly null).

The `getPropertiesOverrides` method returns the properties overrides (possibly null) for this descriptor.

7.4.6 The ActivationGroupDesc.CommandEnvironment Class

The `CommandEnvironment` class allows overriding default system properties and specifying implemention-defined options for an `ActivationGroup`.

```
public static class CommandEnvironment
    implements java.io.Serializable
{
    public CommandEnvironment(String cmdpath, String[] args);
    public boolean equals(java.lang.Object);
    public String[] getCommandOptions();
    public String getCommandPath();
    public int hashCode();
}
```

The constructor creates a CommandEnvironment with the given command, *cmdpath*, and additional command line options, *args*.

The `equals` implements content equality for command environment objects. The `hashCode` method is implemented appropriately so that a CommandEnvironment can be stored in a hash table if necessary.

The `getCommandOptions` method returns the environment object's command line options.

The `getCommandPath` method returns the environment object's command string.

7.4.7 The ActivationGroupID Class

The identifier for a registered activation group serves several purposes:

- it identifies the group uniquely within the activation system, and
- it contains a reference to the group's activation system so that the group can contact its activation system when necessary.

The `ActivationGroupID` is returned from the call to `ActivationSystem.registerGroup` and is used to identify the group within the activation system. This group identifier is passed as one of the arguments to the activation group's special constructor when an activation group is created or re-created.

```
package java.rmi.activation;
public class ActivationGroupID implements java.io.Serializable
{
    public ActivationGroupID(ActivationSystem system);

    public ActivationSystem getSystem();

    public boolean equals(Object obj);

    public int hashCode();
}
```

The `ActivationGroupID` constructor creates a unique group identifier whose `ActivationSystem` is *system*.

The `getSystem` method returns the activation system for the group.

The `hashCode` method returns a hashcode for the group's identifier. Two group identifiers that refer to the same remote group will have the same hash code.

The `equals` method compares two group identifiers for content equality. The method returns true if both of the following conditions are true: 1) the unique identifiers are equivalent (by content), and 2) the activation system specified in each refers to the same remote object.

7.4.8 The ActivationGroup Class

An `ActivationGroup` is responsible for creating new instances of "activatable" objects in its group, informing its `ActivationMonitor` when either: its objects become active or inactive, or the group as a whole becomes inactive.

An `ActivationGroup` is *initially* created in one of several ways:

- as a side-effect of creating a "default" `ActivationDesc` for an object, or
- by an explicit call to the `ActivationGroup.createGroup` method, or
- as a side-effect of activating the first object in a group whose `ActivationGroupDesc` was only registered.

Only the activator can *re-create* an `ActivationGroup`. The activator spawns, as needed, a separate VM (as a child process, for example) for each registered activation group and directs activation requests to the appropriate group. It is implementation specific how VMs are spawned. An activation group is created via the `ActivationGroup.createGroup` static method. The `createGroup` method has two requirements on the group to be created: 1) the group must be a concrete subclass of `ActivationGroup`, and 2) the group must have a constructor that takes two arguments:

- the group's `ActivationGroupID`, and
- the group's initialization data (in a `MarshalledObject`)

When created, the default implementation of `ActivationGroup` will set the system properties to the system properties in force when it `ActivationGroupDesc` was created, and will set the security manager to the `java.rmi.RMISecurityManager`. If your application requires some specific properties to be set when objects are activated in the group, the application should set the properties before creating any `ActivationDescs` (before the default `ActivationGroupDesc` is created).

```
package java.rmi.activation;
public abstract class ActivationGroup
    extends UnicastRemoteObject
    implements ActivationInstantiator
{
    protected ActivationGroup(ActivationGroupID groupID)
        throws java.rmi.RemoteException;

    public abstract MarshalledObject newInstance(ActivationID id,
                                                 ActivationDesc desc)
        throws ActivationException, java.rmi.RemoteException;

    public abstract boolean inactiveObject(ActivationID id)
        throws ActivationException, UnknownObjectException,
               java.rmi.RemoteException;

    public static ActivationGroup createGroup(ActivationGroupID id,
                                             ActivationGroupDesc desc,
                                             long incarnation)
            throws ActivationException;

    public static ActivationGroupID currentGroupID();

    public static void setSystem(ActivationSystem system)
        throws ActivationException;

    public static ActivationSystem getSystem()
        throws ActivationException;
```

```
    protected void activeObject(ActivationID id,
                                java.rmi.MarshalledObject mobj)
        throws ActivationException, UnknownObjectException,
               java.rmi.RemoteException;

    protected void inactiveGroup()
        throws UnknownGroupException, java.rmi.RemoteException;

}
```

The activator calls an activation group's `newInstance` method in order to activate an object with the activation descriptor, *desc*. The activation group is responsible for:

- determining the class for the object using the descriptor's `getClassName` method,
- loading the class from the URL path obtained from the descriptor (using the `getLocation` method),
- creating an instance of the class by invoking the special constructor of the object's class that takes two arguments: the object's `ActivationID`, and a `MarshalledObject` containing the object's initialization data, and
- returning a serialized version of the remote object it just created to the activator.

The method throws `ActivationException` if the instance for the given descriptor could not be created.

The group's `inactiveObject` method is called indirectly via a call to the `Activatable.inactive` method. A remote object implementation must call `Activatable`'s `inactive` method when that object deactivates (the object deems that it is no longer active). If the object does not call `Activatable.inactive` when it deactivates, the object will never be garbage collected since the group keeps strong references to the objects it creates.

The group's `inactiveObject` method unexports the remote object, associated with *id* (only if there are no pending or executing calls to the remote object) from the RMI runtime so that the object can no longer receive incoming RMI calls. If the object currently has pending or executing calls, `inactiveObject` returns *false* and no action is taken.

If the `unexportObject` operation was successful (meaning that the object has no pending or executing calls), the group informs its `ActivationMonitor` (via the monitor's `inactiveObject` method) that the remote object is not currently active so that the remote object will be reactivated by the activator upon a subsequent activation request. If the operation was successful, `inactiveObject` returns *true*. The operation may still succeed if the object is considered active by the `ActivationGroup` but has already been unexported.

The `inactiveObject` method throws an `UnknownObjectException` if the activation group is has no knowledge of this object (e.g., the object was previously reported as inactive, or the object was never activated via the activation group). If the inactive operation fails (e.g., If the remote call to the activator (or activation group) fails, `RemoteException` is thrown.

The `createGroup` method creates and sets the activation group for the current VM. The activation group can only be set if it is not currently set. An activation group is set using the `createGroup` method when the `Activator` initiates the re-creation of an activation group in order to carry out incoming `activate` requests. A group must first register a group descriptor with the `ActivationSystem` before it can be created via this method (passing it the `ActivationID` obtained from previous registration).

The group specified by the `ActivationGroupDesc`, *desc*, must be a concrete subclass of `ActivationGroup` and have a public constructor that takes two arguments; the `ActivationGroupID` for the group and a `MarshalledObject` containing the group's initialization data (obtained from its `ActivationGroupDesc`). Note: if your application creates its own custom activation group, the group must set a security manager in the constructor, or objects cannot be activated in the group.

After the group is created, the `ActivationSystem` is informed that the group is active by calling the `activeGroup` method which returns the `ActivationMonitor` for the group. The application need not call `activeGroup` independently since that call back is taken care of by the `createGroup` method.

Once a group is created, subsequent calls to the `currentGroupID` method will return the identifier for this group until the group becomes inactive, at which point the `currentGroupID` method will return null.

The parameter *incarnation* indicates the current group incarnation, i.e., the number of times the group has been activated. The incarnation number is used as a parameter to the `activeGroup` method, once the group has been successfully created. The incarnation number is zero-based. If the group already exists, or if an error occurs during group creation, the `createGroup` method throws `ActivationException`.

The setSystem method sets the ActivationSystem, *system*, for the VM. The activation system can only be set if no group is currently active. If the activation system is not set via an explicit call to setSystem, then the getSystem method will attempt to obtain a reference to the ActivationSystem by looking up the name java.rmi.activation.ActivationSystem in the Activator's registry. By default, the port number used to look up the activation system is defined by ActivationSystem.SYSTEM_PORT. This port can be overridden by setting the property java.rmi.activation.port. If the activation system is already set when setSystem is called, the method throws ActivationException.

The getSystem method returns the activation system for the VM. The activation system may be set by the setSystem method (described above).

The activeObject method is a protected method used by subclasses to make the activeObject call back to the group's monitor to inform the monitor that the remote object with the specified activation *id* and whose stub is contained in *mobj* is now active. The call is simply forwarded to the group's ActivationMonitor.

The inactiveGroup method is a protected method used by subclasses to inform the group's monitor that the group has become inactive. A subclass makes this call when each object the group participated in activating in the VM has become inactive.

7.4.9 The MarshalledObject Class

A MarshalledObject is a container for an object that allows that object to be passed as a parameter in an RMI call, but postpones deserializing the object at the receiver until the application explicitly requests the object (via a call to the container object). The *serializable* object contained in the MarshalledObject is serialized and deserialized (when requested) with the same semantics as parameters passed in RMI calls, which means that any remote object in the MarshalledObject is represented by a serialized instance of its stub. The object contained by the MarshalledObject may be a remote object, a non-remote object, or an entire graph of remote and non-remote objects.

When an object is placed inside the MarshalledObject wrapper, the serialized form of the object is annotated with the codebase URL (where the class can be loaded); likewise, when the contained object is retrieved from its

`MarshalledObject` wrapper, if the code for the object is not available locally, the URL (annotated during serialization) is used to locate and load the bytecodes for the object's class.

```
package java.rmi;
public final class MarshalledObject implements java.io.Serializable
{
    public MarshalledObject(Object obj)
        throws java.io.IOException;

    public Object get()
        throws java.io.IOException, ClassNotFoundException;

    public int hashCode();

    public boolean equals();
}
```

`MarshalledObject`'s constructor takes a serializable object, *obj*, as its single argument and holds the marshalled representation of the object in a byte stream. The marshalled representation of the object preserves the semantics of objects that are passed in RMI calls:

- each class in the stream is annotated with its codebase URL so that when the object is reconstructed (by a call to the `get` method), the bytecodes for each class can be located and loaded, and
- remote objects are replaced with their proxy stubs.

When an instance of the class `MarshalledObject` is written to a `java.io.ObjectOutputStream`, the contained object's marshalled form (created during construction) is written to the stream; thus, only the byte stream is serialized.

When a `MarshalledObject` is read from a `java.io.ObjectInputStream`, the contained object is not deserialized into a concrete object; the object remains in its marshalled representation until the marshalled object's `get` method is called.

The `get` method always reconstructs a new copy of the contained object from its marshalled form. The internal representation is deserialized with the semantics used for unmarshalling parameters for RMI calls. So, the deserialization of the object's representation loads class code (if not available locally) using the URL annotation embedded in the serialized stream for the object.

The `hashCode` of the marshalled representation of the object is the same as the object passed to the constructor. The `equals` method will return true if the marshalled representation of the objects being compared are equivalent. The comparison that equals uses ignores a class's codebase annotation, meaning that two objects are equivalent if they have the same serialized representation *except* for the codebase of each class in the serialized representation.

Stub/Skeleton Interfaces 8

This section contains the interfaces and classes used by the stubs and skeletons generated by the `rmic` stub compiler.

Topics:

- The RemoteStub Class
- The RemoteCall Interface
- The RemoteRef Interface
- The ServerRef Interface
- The Skeleton Interface
- The Operation Class

8.1 The RemoteStub Class

The `java.rmi.server.RemoteStub` class is the common superclass for stubs of remote objects. Stub objects are surrogates that support exactly the same set of remote interfaces defined by the actual implementation of a remote object.

```
package java.rmi.server;
public abstract class RemoteStub extends java.rmi.RemoteObject {
    protected RemoteStub();
```

```
    protected RemoteStub(RemoteRef ref);
    protected static void setRef(RemoteStub stub, RemoteRef ref);
}
```

The first constructor of `RemoteStub` creates a stub with a null remote reference. The second constructor creates a stub with the given remote reference, *ref*.

The `setRef` method is deprecated (and unsupported) in JDK1.2.

8.1.1 Type Equivalency of Remote Objects with a Stub class

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 non-remote 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.

8.1.2 The Semantics of Object Methods Declared final

The following methods are declared `final` in the `java.lang.Object` class and therefore cannot be overridden by any implementation:

- `getClass`
- `notify`
- `notifyAll`
- `wait`

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

The `wait` and `notify` methods of `java.lang.Object` deal with waiting and notification in the context of the Java language's threading model. While use of these methods for remote stubs 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.

8.2 The RemoteCall Interface

The interface `RemoteCall` is an abstraction used by the stubs and skeletons of remote objects to carry out a call to a remote object.

Note – The `RemoteCall` interface is deprecated in JDK 1.2. The JDK 1.2 stub protocol does not make use of this interface anymore. In JDK 1.2, stubs now use the new `invoke` method which does not require `RemoteCall` as a parameter.

```
package java.rmi.server;
import java.io.*;

public interface RemoteCall {
    ObjectOutput getOutputStream() throws IOException;
    void releaseOutputStream() throws IOException;
    ObjectInput getInputStream() throws IOException;
    void releaseInputStream() throws IOException;
    ObjectOutput getResultStream(boolean success)
        throws IOException, StreamCorruptedException;
    void executeCall() throws Exception;
    void done() throws IOException;
}
```

The method `getOutputStream` returns the output stream into which either the stub marshals arguments or the skeleton marshals results.

The method `releaseOutputStream` releases the output stream; in some transports this will release the stream.

The method `getInputStream` returns the InputStream from which the stub unmarshals results or the skeleton unmarshals parameters.

The method `releaseInputStream` releases the input stream. This will allow some transports to release the input side of a connection early.

The method `getResultStream` returns an output stream (after writing out header information relating to the success of the call). Obtaining a result stream should only succeed once per remote call. If *success* is true, then the result to be marshaled is a normal return; otherwise the result is an exception. `StreamCorruptedException` is thrown if the result stream has already been obtained for this remote call.

The method `executeCall` does whatever it takes to execute the call.

The method `done` allows cleanup after the remote call has completed.

8.3 The RemoteRef Interface

The interface `RemoteRef` represents the handle for a remote object. Each stub contains an instance of RemoteRef that contains the concrete representation of a reference. This remote reference is used to carry out remote calls on the remote object for which it is a reference.

```
package java.rmi.server;

public interface RemoteRef extends java.io.Externalizable {
    Object invoke(Remote obj,
                  java.lang.reflect.Method method,
                  Object[] params,
                  long opnum)
        throws Exception;

    RemoteCall newCall(RemoteObject obj, Operation[] op, int opnum,
        long hash) throws RemoteException;
    void invoke(RemoteCall call) throws Exception;
    void done(RemoteCall call) throws RemoteException;
    String getRefClass(java.io.ObjectOutput out);
    int remoteHashCode();
    boolean remoteEquals(RemoteRef obj);
    String remoteToString();
}
```

The first `invoke` method delegates method invocation to the stub's (*obj*) remote reference and allows the reference to take care of setting up the connection to the remote host, marshaling some representation for the *method* and parameters, *params*, then communicating the method invocation to the remote host. This method either returns the result of the method invocation on the remote object which resides on the remote host or throws a `RemoteException` if the call failed or an application-level exception if the remote invocation throws an exception. Note that the operation number, *opnum*, represents a hash of the method signature and may be used to encode the method for transmission.

The method hash for *opnum* is a 64-bit (long) integer and is computed using the first two 32-bit values of a message digest of a byte stream using the National Institute of Standards and Technology (NIST) Secure Hash Algorithm (SHA-1). The byte stream contains the UTF encoding of a string consisting of the remote method's name followed by its method descriptor (see section 4.3.3 of The Java Virtual Machine Specification for a description of the method descriptor). The hash value is assembled from the first and second 32-bit values of the SHA-1 hash:

```
long hash = sha[1] << 32 + sha[0]
```

Note – The `newCall`, `invoke` and `done` methods are deprecated in JDK 1.2. The stubs generated by rmic using the JDK 1.2 stub protocol version do not use these methods any longer. The sequence of calls consisting of `newCall`, `invoke`, and `done` have been replaced by a new `invoke` method that takes a `Method` object as one of its parameters.

The method `newCall` creates an appropriate call object for a new remote method invocation on the remote object *obj*. The operation array *op* contains the available operations on the remote object. The operation number, *opnum*, is an index into the operation array which specifies the particular operation for this remote call. Passing the operation array and index allows the stubs generator to assign the operation indexes and interpret them. The remote reference may need the operation description to encode in the call.

The method `invoke` executes the remote call. `invoke` will raise any "user" exceptions which should pass through and not be caught by the stub. If any exception is raised during the remote invocation, `invoke` should take care of cleaning up the connection before raising the "user exception" or `RemoteException`.

The method `done` allows the remote reference to clean up (or reuse) the connection. `done` should only be called if the `invoke` call returns successfully (non-exceptionally) to the stub.

The method `getRefClass` returns the nonpackage-qualified class name of the reference type to be serialized onto the stream *out*.

The method `remoteHashCode` returns a hashcode for a remote object. Two remote object stubs that refer to the same remote object will have the same hash code (in order to support remote objects as keys in hashtables). A `RemoteObject` forwards a call to its `hashCode` method to the `remoteHashCode` method of the remote reference.

The method `remoteEquals` compares two remote objects for equality. Two remote objects are equal if they refer to the same remote object. For example, two stubs are equal if they refer to the same remote object. A `RemoteObject` forwards a call to its `equals` method to the `remoteEquals` method of the remote reference.

The method `remoteToString` returns a String that represents the reference of this remote object.

8.4 The ServerRef Interface

The interface `ServerRef` represents the server-side handle for a remote object implementation.

```
package java.rmi.server;

public interface ServerRef extends RemoteRef {

    RemoteStub exportObject(java.rmi.Remote obj, Object data)
        throws java.rmi.RemoteException;

    String getClientHost() throws ServerNotActiveException;
}
```

The method `exportObject` finds or creates a client stub object for the supplied `Remote` object implementation *obj*.The parameter *data* contains information necessary to export the object (such as port number).

The method `getClientHost` returns the host name of the current client. When called from a thread actively handling a remote method invocation, the host name of the client invoking the call is returned. If a remote method call is not currently being service, then `ServerNotActiveException` is called.

8.5 The Skeleton Interface

The interface `Skeleton` is used solely by the implementation of skeletons generated by the `rmic` compiler. A skeleton for a remote object is a server-side entity that dispatches calls to the actual remote object implementation.

Note – The `Skeleton` interfaces is deprecated in JDK1.2. Every 1.1 (and version 1.1 compatible skeletons generated in 1.2 using `rmic -vcompat`, the default) skeleton class generated by the `rmic` stub compiler implements this interface. Skeletons are no longer required for remote method call dispatch in JDK1.2 compatible versions. To generate stubs that are compatible with JDK1.2 or later versions, use the command `rmic` with the option `-v1.2`.

```
package java.rmi.server;

public interface Skeleton {

    void dispatch(Remote obj, RemoteCall call, int opnum, long hash)
        throws Exception;

    Operation[] getOperations();
}
```

The `dispatch` method unmarshals any arguments from the input stream obtained from the *call* object, invokes the method (indicated by the operation number *opnum*) on the actual remote object implementation *obj*, and marshals the return value or throws an exception if one occurs during the invocation.

The `getOperations` method returns an array containing the operation descriptors for the remote object's methods.

8.6 The Operation Class

The class `Operation` holds a description of a Java method for a remote object.

Note – The `Operation` interface is deprecated in JDK 1.2. The JDK 1.2 stub protocol no longer uses the old `RemoteRef.invoke` method which takes an `Operation` as one of its arguments. In JDK 1.2, stubs now use the new `invoke` method which does not require `Operation` as a parameter.

```
package java.rmi.server;

public class Operation {

    public Operation(String op);

    public String getOperation();

    public String toString();
}
```

An `Operation` object is typically constructed with the method signature.

The method `getOperation` returns the contents of the operation descriptor (the value with which it was initialized).

The method `toString` also returns the string representation of the operation descriptor (typically the method signature).

Garbage Collector Interfaces 9

The interfaces and classes in this chapter are used by the distributed garbage collector (DGC)for RMI.

Topics:

- The Interface DGC
- The Lease Class
- The ObjID Class
- The UID Class
- The VMID Class

9.1 The Interface DGC

The DGC abstraction is used for the server side of the distributed garbage collection algorithm. This interface contains the two methods: `dirty` and `clean`. A `dirty` call is made when a remote reference is unmarshaled in a client (the client is indicated by its `VMID`). A corresponding clean call is made when no more references to the remote reference exist in the client. A failed dirty call must schedule a *strong* clean call so that the call's sequence number can be retained in order to detect future calls received out of order by the distributed garbage collector.

A reference to a remote object is *leased* for a period of time by the client holding the reference. The lease period starts when the dirty call is received. It is the client's responsibility to renew the leases, by making additional `dirty` calls, on the remote references it holds before such leases expire. If the client does not renew the lease before it expires, the distributed garbage collector assumes that the remote object is no longer referenced by that client.

```
package java.rmi.dgc;
import java.rmi.server.ObjID;

public interface DGC extends java.rmi.Remote {

    Lease dirty(ObjID[] ids, long sequenceNum, Lease lease)
        throws java.rmi.RemoteException;

    void clean(ObjID[] ids, long seqNum, VMID vmid, boolean strong)
        throws java.rmi.RemoteException;
}
```

The method `dirty` requests leases for the remote object references associated with the object identifiers contained in the array argument *ids*. The *lease* contains a client's unique virtual machine identifier (VMID) and a requested lease period. For each remote object exported in the local virtual machine, the garbage collector maintains a *reference list* — a list of clients that hold references to it. If the lease is granted, the garbage collector adds the client's VMID to the reference list for each remote object indicated in *ids*. The *sequenceNum* parameter is a sequence number that is used to detect and discard late calls to the garbage collector. The sequence number should always increase for each subsequent call to the garbage collector.

Some clients are unable to generate a unique VMID. This is because a VMID is a universally unique identifier only if it contains a *true* host address, an address which some clients are unable to obtain due to security restrictions. In this case, a client can use a VMID of `null`, and the distributed garbage collector will assign a VMID for the client.

The `dirty` call returns a `Lease` object that contains the VMID used and the lease period granted for the remote references. (A server can decide to grant a smaller lease period than the client requests.) A client must use the VMID the garbage collector uses in order to make corresponding `clean` calls when the client drops remote object references.

A client virtual machine need only make one initial `dirty` call for each remote reference referenced in the virtual machine (even if it has multiple references to the same remote object). The client must also make a `dirty` call to renew leases on remote references before such leases expire. When the client no longer has any references to a specific remote object, it must schedule a `clean` call for the object ID associated with the reference.

The `clean` call removes the *vmid* from the reference list of each remote object indicated in *ids*. The sequence number is used to detect late clean calls. If the argument *strong* is true, then the clean call is a result of a failed `dirty` call, and the sequence number for the client *vmid* therefore needs to be remembered.

9.2 The Lease Class

A lease contains a unique virtual machine identifier and a lease duration. A `Lease` object is used to request and grant leases to remote object references.

```
package java.rmi.dgc;

public final class Lease implements java.io.Serializable {

    public Lease(VMID id, long duration);

    public VMID getVMID();

    public long getValue();
}
```

The `Lease` constructor creates a lease with a specific VMID and lease duration. The VMID may be `null`.

The `getVMID` method returns the client VMID associated with the lease.

The `getValue` method returns the lease duration.

9.3 The ObjID Class

The class `ObjID` is used to identify remote objects uniquely in a virtual machine over time. Each identifier contains an object number and an address space identifier that is unique with respect to a specific host. An object identifier is assigned to a remote object when it is exported.

An `ObjID` consists of an object number (a `long`) and a unique identifier for the address space (a `UID`).

```
package java.rmi.server;

public final class ObjID implements java.io.Serializable {

    public ObjID ();

    public ObjID (int num);

    public void write(ObjectOutput out) throws java.io.IOException;

    public static ObjID read(ObjectInput in)
        throws java.io.IOException;

    public int hashCode()

    public boolean equals(Object obj)

    public String toString()
}
```

The first form of the `ObjID` constructor generates a unique object identifier. The second constructor generates *well-known* object identifiers (such as those used by the registry and the distributed garbage collector) and takes as an argument a well-known object number. A well-known object ID generated via this second constructor will not clash with any object IDs generated via the default constructor; to enforce this, the object number of the `ObjID` is set to the "well-known" number supplied in the constructor and all `UID` fields are set to zero.

The method `write` marshals the object ID's representation to an output stream.

The method `read` constructs an object ID whose contents is read from the specified input stream.

The method `hashCode` returns the object number as the hashcode

The `equals` method returns true if *obj* is an `ObjID` with the same contents.

The `toString` method returns a string containing the object ID representation. The address space identifier is included in the string representation only if the object ID is from a non-local address space.

9.4 The UID Class

The class `UID` is an abstraction for creating identifiers that are unique with respect to the host on which it is generated. A `UID` is contained in an `ObjID` as an address space identifier. A UID consists of a number that is unique on the host (an `int`), a time (a `long`), and a count (a `short`).

```
package java.rmi.server;

public final class UID implements java.io.Serializable {

    public UID();

    public UID(short num);

    public int hashCode();

    public boolean equals(Object obj);

    public String toString();

    public void write(DataOutput out) throws java.io.IOException;

   public static UID read(DataInput in) throws java.io.IOException;
}
```

The first form of the constructor creates a pure identifier that is unique with respect to the host on which it is generated. This `UID` is unique under the following conditions: a) the machine takes more than one second to reboot, and b) the machine's clock is never set backward. In order to construct a `UID` that is globally unique, simply pair a `UID` with an `InetAddress`.

The second form of the constructor creates a *well-known* `UID`. There are $2^{16} -1$ such possible well-known IDs. An ID generated via this constructor will not clash with any ID generated via the default `UID` constructor which generates a genuinely unique identifier with respect to this host.

The methods `hashCode`, `equals`, and `toString` are defined for UIDs. Two UIDs are considered equal if they have the same contents.

The method `write` writes the UID to the output stream.

The method `read` constructs a UID whose contents is read from the specified input stream.

9.5 The VMID Class

The class `VMID` provides a universally unique identifier among all Java virtual machines. A `VMID` contains a `UID` and a host address. A `VMID` can be used to identify client virtual machines.

```
package java.rmi.dgc;

public final class VMID implements java.io.Serializable {

    public VMID();

    public static boolean isUnique();

    public int hashCode();

    public boolean equals(Object obj);

    public String toString();
}
```

The `VMID` default constructor creates a globally unique identifier among all Java virtual machines under the following conditions:

- the conditions for uniqueness for objects of the class `java.rmi.server.UID` are satisfied, and
- an address can be obtained for the host that is unique and constant for the lifetime of the `UID` object.

A `VMID` contains the host address of the machine on which it was created. Due to security restrictions, obtaining the true host address is not always possible (for example, the loopback host may be used under security-restricted conditions). The method `isUnique` can be called to determine if `VMIDs` generated in this virtual machine are, in fact, unique among all virtual machines. The method `isUnique` returns true if a valid host name can be determined (other than loopback host); otherwise it returns false.

The `hashCode`, `equals` and `toString` methods are defined for `VMIDs`. Two VMIDs are considered equal if they have the same contents.

RMI Wire Protocol 10

10.1 Overview

The RMI protocol makes use of two other protocols for its on-the-wire format: Java Object Serialization and HTTP. The Object Serialization protocol is used to marshal call and return data. The HTTP protocol is used to “POST” a remote method invocation and obtain return data when circumstances warrant. Each protocol is documented as a separate grammar. Nonterminal symbols in production rules may refer to rules governed by another protocol (either Object Serialization or HTTP). When a protocol boundary is crossed, subsequent productions use that embedded protocol.

Notes about Grammar Notation

- We use a similar notation to that used in the Java Language Specification (see section 2.3 of the JLS).
- Control codes in the stream are represented by literal values expressed in hexadecimal.
- Some nonterminal symbols in the grammar represent application specific values supplied in a method invocation. The definition of such a nonterminal consists of its Java type. A table mapping each of these nonterminals to its respective type follows the grammar.

10.2 RMI Transport Protocol

The wire format for RMI is represented by a *Stream*. The terminology adopted here reflects a client perspective. *Out* refers to output messages and *In* refers to input messages. The contents of the transport header are *not* formatted using Object Serialization.

Stream:
  Out
  In

The input and output streams used by RMI are paired. Each *Out* stream has a corresponding *In* stream. An *Out* stream in the grammar maps to the output stream of a socket (from the client's perspective). An *In* stream (in the grammar) is paired with the corresponding socket's input stream. Since output and input streams are paired, the only header information needed on an input stream is an acknowledgment as to whether the protocol is understood; other header information (such as the magic number and version number) can be implied by the context of stream pairing.

10.2.1 Format of an Output Stream

An output stream in RMI consists of transport *Header* information followed by a sequence of *Messages*. Alternatively, an output stream can contain an invocation embedded in the HTTP protocol.

Out:
  Header Messages
  HttpMessage

Header:
  `0x4a 0x52 0x4d 0x49` *Version Protocol*

Version:
  `0x00 0x01`

Protocol:
  StreamProtocol
  SingleOpProtocol
  MultiplexProtocol

StreamProtocol:
  `0x4b`

SingleOpProtocol:
`0x4c`

MultiplexProtocol:
`0x4d`

Messages:
Message
Messages Message

The *Messages* are wrapped within a particular protocol as specified by *Protocol*. For the *SingleOpProtocol*, there may only be one *Message* after the *Header*, and there is no additional data that the *Message* is wrapped in. The *SingleOpProtocol* is used for invocation embedded in HTTP requests, where interaction beyond a single request and response is not possible.

For the *StreamProtocol* and the *MultiplexProtocol*, the server must respond with a a byte `0x4e` acknowledging support for the protocol, and an *EndpointIdentifier* that contains the host name and port number that the server can see is being used by the client. The client can use this information to determine its host name if it is otherwise unable to do that for security reasons. The client must then respond with another *EndpointIdentifier* that contains the client's default endpoint for accepting connections. This can be used by a server in the *MultiplexProtocol* case to identify the client.

For the *StreamProtocol*, after this endpoint negotiation, the *Messages* are sent over the output stream without any additional wrapping of the data. For the *MultiplexProtocol*, the socket connection is used as the concrete connection for a multiplexed connection, as described in Section 10.6, "RMI's Multiplexing Protocol." Virtual connections initiated over this multiplexed connection consist of a series of *Messages* as described below.

There are three types of output messages: *Call*, *Ping* and *DgcAck*. A *Call* encodes a method invocation. A *Ping* is a transport-level message for testing liveness of a remote virtual machine. A *DGCAck* is an acknowledgment directed to a server's distributed garbage collector that indicates that remote objects in a return value from a server have been received by the client.

Message:
Call
Ping
DgcAck

Call:
`0x50` *CallData*

Ping:
`0x52`

DgcAck:
`0x54` *UniqueIdentifier*

10.2.2 Format of an Input Stream

There are currently three types of input messages: *ReturnData*, *HttpReturn* and *PingAck*. *ReturnData* is the result of a "normal" RMI call. An *HttpReturn* is a return result from an invocation embedded in the HTTP protocol. A *PingAck* is the acknowledgment for a *Ping* message.

In:
ProtocolAck Returns
ProtocolNotSupported
HttpReturn

ProtocolAck:
`0x4e`

ProtocolNotSupported:
`0x4f`

Returns:
Return
Returns Return

Return:
ReturnData
PingAck

ReturnData:
`0x51` *ReturnValue*

PingAck:
`0x53`

10.3 RMI's Use of Object Serialization Protocol

Call and return data in RMI calls are formatted using the Java Object Serialization protocol. Each method invocation's *CallData* is represented by the *ObjectIdentifier* (the target of the call), an *Operation* (a number representing the method to be invoked), a *Hash* (a number that verifies that client stub and remote object skeleton use the same stub protocol), followed by a list of zero or more *Arguments* for the call.

In the JDK1.1 stub protocol the *Operation* represented the method number as assigned by `rmic` and the *Hash* was the stub/skeleton hash which is the stub's interface hash. In the JDK1.2 stub protocol (JDK1.2 stubs are generated using the `-v1.2` option with `rmic`), *Operation* has the value -1 and the *Hash* is a hash representing the method to call. The hash is described in the section "The RemoteRef Interface:.

CallData:
  ObjectIdentifier Operation Hash $Arguments_{opt}$

ObjectIdentifier:
  ObjectNumber UniqueIdentifier

UniqueIdentifier:
  Number Time Count

Arguments:
  Value
  Arguments Value

Value:
  Object
  Primitive

A *ReturnValue* of an RMI call consists of a return code to indicate either a normal or exceptional return, a *UniqueIdentifier* to tag the return value (used to send a DGCAck if necessary) followed by the return result: either the *Value* returned or the *Exception* thrown.

ReturnValue:
  `0x01` *UniqueIdentifier* $Value_{opt}$
  `0x02` *UniqueIdentifier Exception*

Note – *ObjectIdentifier*, *UniqueIdentifier*, and *EndpointIdentifier* are not written out using default serialization, but each uses its own special `write` method (this is not the `writeObject` method used by Object Serialization); the `write` method for each type of identifier adds its component data consecutively to the output stream.

10.3.1 Class Annotation and Class Loading

RMI overrides the `annotateClass` and `resolveClass` methods of `ObjectOutputStream` and `ObjectInputStream` respectively. Each class is annotated with the codebase URL (the location from which the class can be loaded). In the `annotateClass` method, the classloader that loaded the class is queried for its codebase URL. If the classloader is non-null and the classloader has a non-null codebase, then the codebase is written to the stream using the `ObjectOutputStream.writeObject` method; otherwise a null is written to the stream using the `writeObject` method. Note: as an optimization, classes in the "java" package are not annotated, since they are always available to the receiver.

The class annotation is resolved during deserialization using the `ObjectInputStream.resolveClass` method. The `resolveClass` method first reads the annotation via the `ObjectInputStream.readObject` method. If the annotation, a codebase URL, is non-null, then it obtains the classloader for that URL and attempts to load the class. The class is loaded by using a `java.net.URLConnection` to fetch the class bytes (the same mechanism used by a web browser's applet classloader).

10.4 RMI's Use of HTTP POST Protocol

In order to invoke remote methods through a firewall, some RMI calls make use of the HTTP protocol, more specifically HTTP POST. The URL specified in the post header can be one of the following:

```
http://<host>:<port>/
http://<host>:80/cgi-bin/java-rmi?forward=<port>
```

The first URL is used for direct communication with an RMI server on the specific *host* and *port*. The second URL form is used to invoke a "cgi" script on the server which forwards the invocation to the server on the specified *port*.

An *HttpPostHeader* is a standard HTTP header for a POST request. An *HttpResponseHeader* is a standard HTTP response to a post. If the response status code is not 200, then it is assumed that there is no *Return*. Note that only a single RMI call is embedded in an HTTP POST request.

HttpMessage:
HttpPostHeader Header Message

HttpReturn:
HttpResponseHeader Return

Note – Only the *SingleOpProtocol* appears in the *Header* of an *HttpMessage*. An HttpReturn does not contain a protocol acknowledgment byte.

10.5 Application Specific Values for RMI

This table lists the nonterminal symbols that represent application specific values used by RMI. The table maps each symbol to its respective type. Each is formatted using the protocol in which it is embedded.

Count	`short`
Exception	`java.lang.Exception`
Hash	`long`
Hostname	`UTF`
Number	`int`
Object	`java.lang.Object`
ObjectNumber	`long`
Operation	`int`
PortNumber	`int`
Primitive	`byte, int, short, long...`
Time	`long`

10.6 RMI's Multiplexing Protocol

The purpose of multiplexing is to provide a model where two endpoints can each open multiple full duplex connections to the other endpoint in an environment where only one of the endpoints is able to open such a bidirectional connection using some other facility (e.g., a TCP connection). RMI use this simple multiplexing protocol to allow a client to connect to an RMI server object in some situations where that is otherwise not possible. For example, some security managers for applet environments disallow the creation of server sockets to listen for incoming connections, thereby preventing such applets to export RMI objects and service remote calls from direct socket connections. If the applet *can* open a normal socket connection to its codebase host, however, then it can use the multiplexing protocol over that connection to allow the codebase host to invoke methods on RMI objects exported by the applet. This section describes how the format and rules of the multiplexing protocol.

10.6.1 Definitions

This sections defines some terms as they are used in the rest of the description of the protocol.

An *endpoint* is one of the two users of a connection using the multiplexing protocol.

The multiplexing protocol must layer on top of one existing bidirectional, reliable byte stream, presumably initiated by one of the endpoints to the other. In current RMI usage, this is always a TCP connection, made with a java.net.Socket object. This connection will be referred to as the *concrete connection*.

The multiplexing protocol facilitates the use of *virtual connections*, which are themselves bidirectional, reliable byte streams, representing a particular session between two endpoints. The set of virtual connections between two endpoints over a single concrete connection comprises a *multiplexed connection*. Using the multiplexing protocol, virtual connections can be opened and closed by either endpoint. The state of an virtual connection with respect to a given endpoint is defined by the elements of the multiplexing protocol that are sent and received over the concrete connection. Such state involves if the connection is open or closed, the actual data that has been transmitted across, and the related flow control mechanisms. If not otherwise qualified, the term *connection* used in the remainder of this section means *virtual connection*.

A virtual connections within a given multiplexed connection is identified by a 16 bit integer, known as the *connection identifier*. Thus, there exist 65,536 possible virtual connections in one multiplexed connection. The implementation may limit the number of these virtual connections that may be used simultaneously.

10.6.2 Connection State and Flow Control

Connections are manipulated using the various *operations* defined by the multiplexing protocol. The following are the names of the operations defined by the protocol: OPEN, CLOSE, CLOSEACK, REQUEST, and TRANSMIT. The exact format and rules for all the operations are detailed in Section 10.6.3, "Protocol Format."

The OPEN, CLOSE, and CLOSEACK operations control connections becoming opened and closed, while the REQUEST and TRANSMIT operations are used to transmit data across an open connection within the constraints of the flow control mechanism.

Connection States

A virtual connection is *open* with respect to a particular endpoint if the endpoint has sent an OPEN operation for that connection, or it has received an OPEN operation for that connection (and it had not been subsequently closed). The various protocol operations are described below.

A virtual connection is *pending close* with respect to a particular endpoint if the endpoint has sent a CLOSE operation for that connection, but it has not yet received a subsequent CLOSE or CLOSEACK operation for that connection.

A virtual connection is *closed* with respect to a particular endpoint if it has never been opened, or if it has received a CLOSE or a CLOSEACK operation for that connection (and it has not been subsequently opened).

Flow Control

The multiplexing protocol using a simple packeting flow control mechanism to allow multiple virtual connections to exist in parallel over the same concrete connection. The high level requirement of the flow control mechanism is that the state of all virtual connections is independent; the state of one connection may not affect the behavior of others. For example, if the data buffers handling data coming in from one connection become full, this cannot prevent the transmission and processing of data for any other connection. This is necessary if the proceedings of one connection is dependent on the completion of the use of another connection, such as would happen with recursive RMI calls. Therefore, the practical implication is that the implementation must always be able to consume and process all of the multiplexing protocol data ready for input on the concrete connection (assuming that it conforms to this specification).

Each endpoint has two state values associated with each connection: how many bytes of data the endpoint has requested but not received (*input request count*) and how many bytes the other endpoint has requested but have not been supplied by this endpoint (*output request count*).

An endpoint's output request count is increased when it receives a REQUEST operation from the other endpoint, and it is decreased when it sends a TRANSMIT operation. An endpoint's input request count is increased when it sends a REQUEST operation, and it is decreased when it receives a TRANSMIT operation. It is a protocol violation if either of these values becomes negative.

It is a protocol violation for an endpoint to send a REQUEST operation that would increase its input request count to more bytes that it can currently handle without blocking. It should, however, make sure that its input request count is greater than zero if the user of the connection is waiting to read data.

It is a protocol violation for an endpoint to send a TRANSMIT operation containing more bytes that its output request count. It may buffer outgoing data until the user of the connection requests that data written to the connection be explicitly flushed. If data must be sent over the connection, however, by either an explicit flush or because the implementation's output buffers are full, then the user of the connection may be blocked until sufficient TRANSMIT operations can proceed.

Beyond the rules outlined above, implementations are free to send REQUEST and TRANSMIT operations as deemed appropriate. For example, an endpoint may request more data for a connection even if its input buffer is not empty.

10.6.3 Protocol Format

The byte stream format of the multiplexing protocol consists of a contiguous series of variable length records. The first byte of the record is an operation code that identifies the operation of the record and determines the format of the rest of its content. The following legal operation codes are defined:

value	name
0xE1	OPEN
0xE2	CLOSE
0xE3	CLOSEACK
0xE4	REQUEST
0xE5	TRANSMIT

It is a protocol violation if the first byte of a record is not one of the defined operation codes. The following sections describe the format of the records for each operation code.

OPEN operation

This is the format for records of the OPEN operation:

size (bytes)	name	description
1	*opcode*	operation code (OPEN)
2	*ID*	connection identifier

An endpoint sends an OPEN operation to open the indicated connection. It is a protocol violation if *ID* refers to a connection that is currently open or pending close with respect to the sending endpoint. After the connection is opened, both input and request count states for the connection are zero for both endpoints.

Receipt of an OPEN operation indicates that the other endpoint is opening the indicated connection. After the connection is opened, both input and output request count states for the connection are zero for both endpoints.

To prevent identifier collisions between the two endpoints, the space of valid connection identifiers is divided in half, depending on the value of the most significant bit. Each endpoint is only allowed to open connections with a particular value for the high bit. The endpoint that initiated the concrete connection must only open connections with the high bit set in the identifier and the other endpoint must only open connections with a zero in the high bit. For example, if an RMI applet that cannot create a server socket initiates a multiplexed connection to its codebase host, the applet may open virtual connections in the identifier range 0x8000-7FFF, and the server may open virtual connection in the identifier range 0-0x7FFF.

CLOSE operation

This is the format for records of the CLOSE operation:

size (bytes)	name	description
1	*opcode*	operation code (OPEN)
2	*ID*	connection identifier

An endpoint sends a CLOSE operation to close the indicated connection. It is a protocol violation if *ID* refers to a connection that is currently closed or pending close with respect to the sending endpoint (it may be pending close with respect to the receiving endpoint if it has also sent a CLOSE operation for this connection). After sending the CLOSE, the connection becomes pending close for the sending endpoint. Thus, it may not reopen the connection until it has received a CLOSE or a CLOSEACK for it from the other endpoint.

Receipt of a CLOSE operation indicates that the other endpoint has closed the indicated connection, and it thus becomes closed on the receiving endpoint. Although the receiving endpoint may not send any more operations for this connection (until it is opened again), it still should provide data in the implementation's input buffers to readers of the connection. If the connection had previously been open instead of pending close, the receiving endpoint must respond with a CLOSEACK operation for the connection.

CLOSEACK operation

The following is the format for records with the CLOSEACK operation:

size (bytes)	name	description
1	*opcode*	operation code (OPEN)
2	*ID*	connection identifier

An endpoint sends a CLOSEACK operation to acknowledge a CLOSE operation from the receiving endpoint. It is a protocol violation if ID refers to a connection that is not pending close for the receiving endpoint when the operation is received.

Receipt of a CLOSEACK operation changes the state of the indicated connection from pending close to closed, and thus the connection may be reopened in the future.

REQUEST operation

This is the format for records of the REQUEST operation:

size (bytes)	name	description
1	*opcode*	operation code (OPEN)
2	*ID*	connection identifier
4	*count*	number of additional bytes requested

An endpoint sends a REQUEST operation to increase its input request count for the indicated connection. It is a protocol violation if *ID* does not refer to a connection that is open with respect to the sending endpoint. The endpoint's input request count is incremented by the value *count*. The value of *count* is a signed 32 bit integer, and it is a protocol violation if it is negative or zero.

Receipt of a REQUEST operation causes the output request count for the indicated connection to increase by *count*. If the connection is pending close by the receiving endpoint, then any REQUEST operations may be ignored.

TRANSMIT operation

This is the format for records of the TRANSMIT operation.

size (bytes)	name	description
1	*opcode*	operation code (OPEN)
2	*ID*	connection identifier
4	*count*	number of bytes in transmission
count	data	transmission data

An endpoint sends a TRANSMIT operation to actually transmit data over the indicated connection. It is a protocol violation if ID does not refer to a connection that is open with respect to the sending endpoint. The endpoint's output request count is decremented by the value *count*. The value of *count* is a signed 32 bit integer, and it is a protocol violation if it is negative or zero. It is also a protocol violation if the TRANSMIT operation would cause the sending endpoint's output request count to become negative.

Receipt of a TRANSMIT operation causes the count bytes of data to be added to the queue of bytes available for reading from the connection. The receiving endpoint's input request count is decremented by *count*. If this causes the input request count to become zero and the user of the connection is trying to read more data, the endpoint should respond with another REQUEST operation. If the connection is pending close by the receiving endpoint, then any TRANSMIT operations may be ignored.

Protocol Violations

If a protocol violation occurs, as defined above or if a communication error is detected in the concrete connection, then the multiplexed connection is *shut down*. The real connection is terminated, and all virtual connections become closed immediately. Data already available for reading from virtual connections may be read by the users of the connections.

Exceptions In RMI

A

Topics:

- Exceptions During Remote Object Export
- Exceptions During RMI Call
- Exceptions or Errors During Return
- Naming Exceptions
- Other Exceptions

A.1 Exceptions During Remote Object Export

When a remote object class is created that extends `UnicastRemoteObject`, the object is exported, meaning it can receive calls from external Java virtual machines and can be passed in an RMI call as either a parameter or return value. An object can either be exported on an anonymous port or on a specified port. For objects not extended from `UnicastRemoteObject`, the `java.rmi.server.UnicastRemoteObject.exportObject` method is used to explicitly export the object.

Exception	Context
java.rmi.StubNotFoundException	1. Class of stub not found. 2. Name collision with class of same name as stub causes one of these errors: • Stub can't be instantiated. • Stub not of correct class. 3. Bad URL due to wrong codebase. 4. Stub not of correct class.
java.rmi.server.SkeletonNotFoundException *note: this exception is deprecated in JDK1.2*	1. Class of skeleton not found. 2. Name collision with class of same name as skeleton causes one of these errors: • Skeleton can't be instantiated. • Skeleton not of correct class. 3. Bad URL due to wrong codebase. 4. Skeleton not of correct class.
java.rmi.server.ExportException	The port is in use by another VM.

A.2 Exceptions During RMI Call

Exception	Context
java.rmi.UnknownHostException	Unknown host.
java.rmi.ConnectException	Connection refused to host.
java.rmi.ConnectIOException	I/O error creating connection.
java.rmi.MarshalException	I/O error marshaling transport header, marshaling call header, or marshaling arguments.
java.rmi.NoSuchObjectException	Attempt to invoke a method on an object that is no longer available.
java.rmi.StubNotFoundException	Remote object not exported.
java.rmi.activation.ActivateFailedException	Thrown by RMI runtime when activation fails during a remote call to an activatable object

A.3 Exceptions or Errors During Return

Exception	Context
java.rmi.UnmarshalException	1. Corrupted stream leads to either an I/O or protocol error when: • Marshaling return header. • Checking return type. • Checking return code. • Unmarshaling return. 2. Return value class not found.
java.rmi.UnexpectedException	An exception not mentioned in the method signature occurred (excluding runtime exceptions). The `UnexpectedException` exception object contains the underlying exception that was thrown by the server.

Exception	Context
java.rmi.ServerError	Any error that occurs while the server is executing a remote method. The `ServerError` exception object contains the underlying error that was thrown by the server,
java.rmi.ServerException	Any remote exception that occurs while the server is executing a remote method. For examples, see Section A.3.1, "Possible Causes of java.rmi.ServerException".
java.rmi.ServerRuntimeException *note: this exception is deprecated in JDK1.2*	This exception is not thrown by servers running JDK1.2 compatible versions. A `RuntimeException` are propagated to clients in tact.

A.3.1 Possible Causes of java.rmi.ServerException

These are some of the underlying exceptions which can occur on the server when the server is itself executing a remote method invocation. These exceptions are wrapped in a `java.rmi.ServerException`; that is the `java.rmi.ServerException` contains the original exception for the client to extract. These exceptions are wrapped by `ServerException` so that the client will know that its own remote method invocation on the server did not fail, but that a secondary remote method invocation made by the server failed.

Exception	Context
java.rmi.server.SkeletonMismatchException *note: this exception is deprecated in JDK1.2*	Hash mismatch of stub and skeleton.
java.rmi.UnmarshalException	I/O error unmarshaling call header. I/O error unmarshaling arguments.
java.rmi.MarshalException	Protocol error marshaling return.
java.rmi.RemoteException	Method number out of range due to corrupted stream.

A.4 Naming Exceptions

The following table lists the exceptions specified in methods of the `java.rmi.Naming` class and the `java.rmi.registry.Registry` interface.

Exception	Context
java.rmi.AccessException	Operation disallowed. The registry restricts bind, rebind, and unbind to the same host. The lookup operation can originate from any host.
java.rmi.AlreadyBoundException	Attempt to bind a name that is already bound.
java.rmi.NotBoundException	Attempt to look up a name that is not bound.
java.rmi.UnknownHostException	Attempt to contact a registry on an unknown host.

A.5 Activation Exceptions

The following table lists the exceptions that can be thrown in activities involving activatable objects. The activation API is in the package `java.rmi.activation`.

Exception	Context
java.rmi.activation.ActivateFailedException	Thrown by RMI runtime when activation fails during a remote call to an activatable object.
java.rmi.activation.ActivationException	General exception class used by the activation interfaces and classes.
java.rmi.activation.UnknownGroupException	Thrown by methods of the activation classes and interfaces when the ActivationGroupID parameter or ActivationGroupID in an ActivationGroupDesc parameter is invalid.
java.rmi.activation.UnknownObjectException	Thrown by methods of the activation classes and interfaces when the ActivationID parameter is invalid.

A.6 Other Exceptions

Exception	Context
java.rmi.RMISecurityException *note: this exception is deprecated in JDK1.2*	A security exception that is thrown by the RMISecurityManager.
java.rmi.server.ServerCloneException	Clone failed.
java.rmi.server.ServerNotActiveException	Attempt to get the client host via the RemoteServer.getClientHost method when the remote server is not executing in a remote method.
java.rmi.server.SocketSecurityException	Attempt to export object on an illegal port.

Properties In RMI B

Topics:

- Server Properties
- Activation Properties
- Other Properties

B.1 Server Properties

The following table contains a list of properties typically used by servers for configuration. Note that properties are typically restricted from being set from applets.

Property	Description
java.rmi.server.codebase	Indicates the codebase URL of classes originating from the VM. The codebase property is used to annotate class descriptors of classes *originating* from a VM so that the class for an object sent as a parameter or return value in a remote method call can be loaded at the receiver.
java.rmi.server.disableHttp	If set to true, disables the use of HTTP for RMI calls. This means that RMI will never resort to using HTTP to invoke a call via a firewall. Defaults to false (HTTP usage is enabled).
java.rmi.server.hostname	RMI uses IP addresses to indicate the location of a server (embedded in a remote reference). If the use of a hostname is desired, this property is used to specify the fully-qualified hostname for RMI to use for remote objects exported to the local VM. The property can also be set to an IP address. Not set by default.
java.rmi.dgc.leaseValue	Sets the lease duration that the RMI runtime grants to clients referencing remote objects in the VM. Defaults to 10 minutes.

Property	Description
java.rmi.server.logCalls	If set to true, server call logging is turned on and prints to stderr. Defaults to false.
java.rmi.server.useCodebaseOnly	If set to true, when RMI loads classes (if not available via CLASSPATH) they are only loaded using the URL specified by the property java.rmi.server.codebase.
java.rmi.server.useLocalHostname	If the `java.rmi.server.hostname` property is not set and this property is set, then RMI will not use an IP address to denote the location (embedded in remote references) of remote objects that are exported into the VM. Instead, RMI will use the value of the call to the method `java.net.InetAddress.getLocalHost`.

B.2 Activation Properties

The following table contains a list of properties used in activation.

Property	Description
java.rmi.activation.port	The port number on which the `ActivationSystem` is exported. This port number should be specified in a VM if the activation daemon **`rmid`** uses a port other than the default.
java.rmi.activation.activator.class	The class that implements the interface `java.rmi.activation.Activator`. This property is used internally to locate the resident implementation of the Activator from which the stub class name can be found.

B.3 Other Properties

These properties are used to locate specific implementation classes within implementation packages. Note: all these properties have been deprecated in JDK1.2.

Property	**Description**
java.rmi.loader.packagePrefix (deprecated in JDK1.2)	The package prefix for the class that implements the interface java.rmi.server.LoaderHandler. Defaults to sun.rmi.server.
java.rmi.registry.packagePrefix (deprecated in JDK1.2)	The package prefix for the class that implements the interface java.rmi.registry.RegistryHandler. Defaults to sun.rmi.registry.
java.rmi.server.packagePrefix (deprecated in JDK1.2)	The server package prefix. Assumes that the implementation of the server reference classes (such as UnicastRef and UnicastServerRef) are located in the package defined by the prefix. Defaults to sun.rmi.server.