

JavaTM Remote Method Invocation Specification

JavaTM Remote Method Invocation (RMI) is a distributed object model for the Java programming language that retains the semantics of the Java platform's 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 the Java platform.

Revision 1.10, JavaTM 2 SDK, Standard Edition, v1.5.0

Copyright (c) 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers. Portions Copyright Eastman Kodak Company 1992. This product includes code licensed from RSA Data Security. This product includes software developed by the Apache Software Foundation (<http://www.apache.org>). Portions licensed from Taligent, Inc. This software is based in part on the work of the Independent JPEG Group. The Graphics Interchange Format is the Copyright property of CompuServe Incorporated.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, J2SE, the Duke logo, the Java Coffee Cup logo and the Solaris logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. Lucida is a registered trademark or trademark of Bigelow & Holmes in the U.S. And other countries. GIF(sm) is a Service Mark property of CompuServe Incorporated.

Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright (c) 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, E'tats-Unis. Tous droits re'serve's.

Sun Microsystems, Inc. de'tient les droits de proprie'te' intellectuelle sur la technologie incluse dans le produit de'crit dans ce document. En particulier, et de manie're non-exclusive, ces droits de proprie'te' intellectuelle peuvent inclure un ou plusieurs brevets U.S. e'nume're's a' l'adresse suivante : <http://www.sun.com/patents> et un ou plusieurs brevets suppl'e'mentaires ou une ou plusieurs demandes de brevet pendantes aux E'tats-Unis et dans d'autres pays.

Ce document et le produit auquel il se rapporte sont distribue's en vertu de licences limitant leur utilisation, leur copie, leur diffusion et leur de'compilation. Aucune partie de ce produit ou de ce document ne peut e'^tre reproduite, sous quelque forme et par quelque moyen que ce soit, sans l'autorisation e'crite pre'alable de Sun et, le cas e'che'ant, de ses conce'dants.

Tout logiciel de tiers, y compris toute technologie de police, est prote'ge' par droit d'auteur et obtenu sous licence aupre's de fournisseurs de Sun. Certaines parties sont prote'ge'es par droit d'auteur Eastman Kodak Company 1992. Ce produit inclut un code conce'de' sous licence par RSA Data Security. Ce produit inclut un logiciel de'veloppe' par Apache Software Foundation (<http://www.apache.org>). Certaines parties sont obtenues sous licence de Taligent, Inc. Ce logiciel repose en partie sur le travail du JPEG Independant Group. Le Format Graphics Interchange est la proprie'te' prote'ge'e par droit d'auteur de CompuServe Incorporated.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, J2SE, le logo Duke, le logo Java Coffee Cup et le logo Solaris sont des marques ou des marques de'pose'es de Sun Microsystems, Inc. aux E'tats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilise'es sous licence et sont des marques ou des marques de'pose'es de SPARC International, Inc. aux E'tats-Unis et dans d'autres pays. Les produits portant les marques SPARC reposent sur l'architecture de'veloppe'e par Sun Microsystems, Inc. UNIX est une marque de'pose'e aux E'tats-Unis et dans d'autres pays, conce'de'e sous licence exclusivement par l'interme'diaire de X/Open Company, Ltd. Lucida est une marque de'pose'e ou une marque de Bigelow & Holmes aux E'tats-Unis et dans d'autres pays. GIF(sm) est une proprie'te' de marque de service de CompuServe Incorporated.

LA DOCUMENTATION EST FOURNIE " TELLE QUELLE " ET TOUTES CONDITIONS, DE'CLARATIONS ET GARANTIES EXPRESSES OU TACITES, Y COMPRIS TOUTE GARANTIE TACITE DE QUALITE' MARCHANDE, D'ADE'QUATION A' UN USAGE PARTICULIER OU D'ABSENCE DE CONTREFAC,ON DE DROITS DE TIERS, SONT REJETE'ES, EXCEPTE' DANS LE CAS OU' L'EXCLUSION OU LA LIMITATION DE TELLES GARANTIES NE SERAIT PAS AUTORISE'E PAR LA LE'GISLATION EN VIGUEUR.

Table of Contents



1 Introduction	1
1.1 Background	1
1.2 System Goals	2
2 Distributed Object Model	3
2.1 Distributed Object Applications	3
2.2 Definition of Terms	5
2.3 The Distributed and Nondistributed Models Contrasted	5
2.4 Overview of RMI Interfaces and Classes	6
2.5 Implementing a Remote Interface	9
2.6 Parameter Passing in Remote Method Invocation	10
2.7 Locating Remote Objects	12
3 RMI System Overview	15
3.1 Stubs and Skeletons	15
3.2 Thread Usage in Remote Method Invocations	16
3.3 Garbage Collection of Remote Objects	16



3.4	Dynamic Class Loading.....	18
3.5	RMI Through Firewalls Via Proxies.....	19
4	Client Interfaces	23
4.1	The Remote Interface.....	23
4.2	The RemoteException Class	23
4.3	The Naming Class.....	23
5	Server Interfaces	25
5.1	The RemoteObject Class.....	26
5.2	The RemoteServer Class.....	26
5.3	The UnicastRemoteObject Class.....	26
5.4	The Unreferenced Interface.....	29
5.5	The RMISecurityManager Class.....	30
5.6	The RMIClassLoader Class	30
5.7	The LoaderHandler Interface.....	30
5.8	RMI Socket Factories	30
5.9	The RMIFailureHandler Interface.....	33
5.10	The LogStream Class	34
5.11	Stub and Skeleton Compiler	34
6	Registry Interfaces	35
6.1	The Registry Interface	35
6.2	The LocateRegistry Class	36
6.3	The RegistryHandler Interface	37
7	Remote Object Activation	39
7.1	Overview.....	39



7.2	Activation Protocol	41
7.3	Implementation Model for an “Activatable” Remote Object	42
7.4	Activation Interfaces	55
8	Stub/Skeleton Interfaces	73
8.1	The RemoteStub Class	73
8.2	The RemoteCall Interface	75
8.3	The RemoteRef Interface	76
8.4	The ServerRef Interface	79
8.5	The Skeleton Interface	79
8.6	The Operation Class	80
9	Garbage Collector Interfaces.	83
9.1	The DGC Interface	83
9.2	The Lease Class	83
9.3	The ObjID Class	83
9.4	The UID Class	84
9.5	The VMID Class	84
10	RMI Wire Protocol	85
10.1	Overview	85
10.2	RMI Transport Protocol	86
10.3	RMI’s Use of Object Serialization Protocol	89
10.4	RMI’s Use of HTTP POST Protocol	91
10.5	Application-Specific Values for RMI	92
10.6	RMI’s Multiplexing Protocol	93
A	Exceptions In RMI	101



A.1	Exceptions During Remote Object Export.....	102
A.2	Exceptions During RMI Call.....	103
A.3	Exceptions or Errors During Return	104
A.4	Naming Exceptions	105
A.5	Activation Exceptions	106
A.6	Other Exceptions	107
B	Properties In RMI.....	109
B.1	Server Properties	110
B.2	Activation Properties.....	112
B.3	Other Properties	113

Introduction



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™ programming language supports sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone.

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

RPC, however, does not translate well into distributed object systems, where communication between program-level *objects* residing in different address spaces is needed. In order to match the semantics of object invocation,

distributed object systems require *remote method invocation* or RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

The Java platform's remote method invocation system described in this specification has been specifically designed to operate in the Java application environment. The Java programming language's RMI system assumes the homogeneous environment of the Java virtual machine (JVM), and the system can therefore take advantage of the Java platform's object model whenever possible.

1.2 System Goals

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

- Support seamless remote invocation on objects in different virtual machines
- Support callbacks from servers to applets
- Integrate the distributed object model into the Java programming language in a natural way while retaining most of the Java programming language's object semantics
- Make differences between the distributed object model and local Java platform's object model apparent
- Make writing reliable distributed applications as simple as possible
- Preserve the type-safety provided by the Java platform's runtime environment
- Support various reference semantics for remote objects; for example live (nonpersistent) references, persistent references, and lazy activation
- Maintain the safe environment of the Java platform provided by security managers and class loaders

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

The first two chapters in this specification describe the distributed object model for the Java programming language and the system overview. The remaining chapters describe the RMI client and server visible APIs which are part of the Java 2 platform.

Distributed Object Model



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 application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to:

- Locate remote objects

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

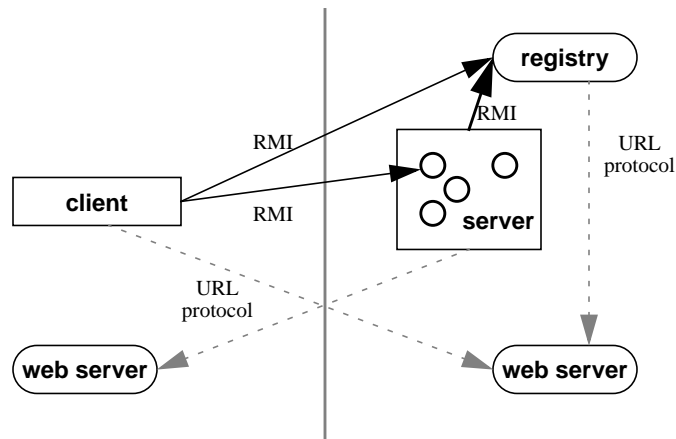
- Communicate with remote objects

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

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

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

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



2.2 Definition of Terms

In the Java platform's distributed object model, a *remote object* is one whose methods can be invoked from another Java virtual machine, potentially on a different host. An object of this type is described by one or more *remote interfaces*, which are interfaces written in the Java programming language that declare the methods of the remote object.

Remote method invocation (RMI) is the action of invoking a method of a remote interface on a remote object. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

2.3 The Distributed and Nondistributed Models Contrasted

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

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

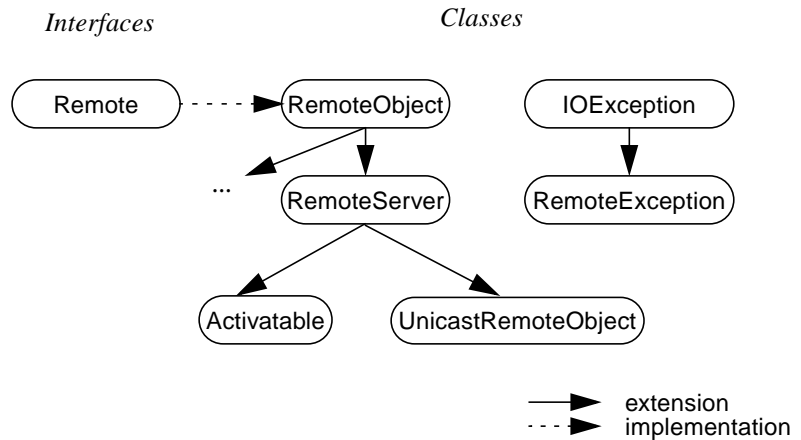
The Java platform's distributed object model differs from the Java platform's object model in these ways:

- Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces.
- Non-remote arguments to, and results from, a remote method invocation are passed by copy rather than by reference. This is because references to objects are only useful within a single virtual machine.
- A remote object is passed by reference, not by copying the actual remote implementation.
- The semantics of some of the methods defined by class `java.lang.Object` are specialized for remote objects.

- Since the failure modes of invoking remote objects are inherently more complicated than the failure modes of invoking local objects, clients must deal with additional exceptions that can occur during a remote method invocation.

2.4 Overview of RMI Interfaces and Classes

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



2.4.1 The `java.rmi.Remote` Interface

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

- A remote interface must at least extend, either directly or indirectly, the interface `java.rmi.Remote`.
- Each method declaration in a remote interface or its super-interfaces must satisfy the requirements of a *remote method* declaration as follows:
 - A remote method declaration must include the exception `java.rmi.RemoteException` (or one of its superclasses such as `java.io.IOException` 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 object that is *serializable*. This includes primitive types, remote objects, and non-remote objects that implement the `java.io.Serializable` interface. For more details on how to make classes serializable, see the “Java Object Serialization Specification.” Classes, for parameters or return values, that are not available locally are downloaded dynamically by the RMI system. See the section on “Dynamic Class Loading” for more information on how RMI downloads parameter and return value classes when reading parameters, return values and exceptions.

2.6.1 *Passing Non-remote Objects*

A non-remote object, that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation, is passed by *copy*; that is, the object is serialized using the object serialization mechanism of the Java platform.

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

When a non-remote object is returned from a remote method invocation, a new object is created in the calling virtual machine.

2.6.2 Passing Remote Objects

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

2.6.3 Referential Integrity

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

2.6.4 Class Annotation

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

2.6.5 Parameter Transmission

Parameters in an RMI call are written to a stream that is a subclass of the class `java.io.ObjectOutputStream` in order to serialize the parameters to the destination of the remote call. The `ObjectOutputStream` subclass overrides the `replaceObject` method to replace each exported remote object with its corresponding stub instance. Parameters that are objects are written to the stream using the `ObjectOutputStream`'s `writeObject` method. The `ObjectOutputStream` calls the `replaceObject` method for each object

written to the stream via the `writeObject` method (that includes objects referenced by those objects that are written). The `replaceObject` method of RMI's subclass of `ObjectOutputStream` returns the following:

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

RMI's subclass of `ObjectOutputStream` also implements the `annotateClass` method that annotates the call stream with the location of the class so that it can be downloaded at the receiver. See the section “Dynamic Class Loading” 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



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 responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements.

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

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

- returns the value to the caller.

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

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

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

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

3.2 Thread Usage in Remote Method Invocations

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

3.3 Garbage Collection of Remote Objects

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

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

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine’s garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine’s garbage collector in the usual ways by holding normal or weak references to objects.

As long as a local reference to a remote object exists, it cannot be garbage-collected and it can be passed in remote calls or returned to clients. Passing a remote object adds the identifier for the virtual machine to which it was passed to the referenced set. A remote object needing unreferenced notification must implement 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 JVM, class definitions are required for all of the types of objects in the stream. The unmarshalling process first attempts to resolve classes by name in its local class loading context (the context class loader of the current thread). RMI also provides a facility for dynamically loading the class definitions for the actual types of objects passed as parameters and return values for remote method invocations from network locations specified by the transmitting endpoint. This includes the dynamic downloading of remote stub classes corresponding to particular remote object implementation classes (and used to contain remote references) as well as any other type that is passed by value in RMI calls, such as the subclass of a declared parameter type, that is not already available in the class loading context of the unmarshalling side.

To support dynamic class loading, the RMI runtime uses special subclasses of `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` for the marshal streams that it uses for marshalling and unmarshalling RMI parameters and return values. These subclasses respectively override the `annotateClass` method of `ObjectOutputStream` and 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 class loading in RMI.

3.5 *RMI Through Firewalls Via Proxies*

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

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

3.5.1 *How an RMI Call is Packaged within the HTTP Protocol*

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

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 is forwarded to the HTTP server listening on port 80 of the host machine, and a CGI script is executed to forward the call to the target RMI server port on the same machine.

3.5.2 *The Default Socket Factory*

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

- Client sockets first attempt a direct socket connection. Client sockets automatically attempt HTTP connections to hosts that cannot be contacted with a direct socket if that direct socket connection results in either a `java.net.NoRouteToHostException` or a `java.net.UnknownHostException` being thrown. If a direct socket connection results in any other `java.io.IOException` being thrown, such as a `java.net.ConnectException`, the implementation may attempt an HTTP connection.
- Server sockets automatically detect if a newly-accepted connection is an HTTP POST request, and if so, return a socket that will expose only the body of the request to the transport and format its output as an HTTP response.

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

3.5.3 *Configuring the Client*

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

3.5.4 *Configuring the Server*

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

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

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

For example, use this command to start the RMI server class `ServerImpl` on the machine `chatsubo.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 interpreter for the Java programming language to execute a class internal to the transport layer which forwards the request to the appropriate RMI server port.
- Defines properties in the Java virtual machine with the same names and values as the CGI 1.0 defined environment variables.

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

3.5.5 Performance Issues and Limitations

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

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

Client Interfaces



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

Topics:

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

4.1 The Remote Interface

See the `Remote` API documentation, and for more details on how to define a remote interface see the section “The `java.rmi.Remote` Interface”.

4.2 The RemoteException Class

See the `RemoteException` API documentation.

4.3 The Naming Class

See the `Naming` API documentation.

Server Interfaces

5 

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

Topics:

- The `RemoteObject` Class
- The `RemoteServer` Class
- The `UnicastRemoteObject` Class
- The `Unreferenced` Interface
- The `RMI SecurityManager` Class
- The `RMIClassLoader` Class
- The `LoaderHandler` Interface
- RMI Socket Factories
- The `RMI FailureHandler` Interface
- The `LogStream` Class
- Stub and Skeleton Compiler

5.1 *The RemoteObject Class*

See the `RemoteObject` API documentation.

5.2 *The RemoteServer Class*

See the `RemoteServer` API documentation.

5.3 *The UnicastRemoteObject Class*

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

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

```
package java.rmi.server;

public class UnicastRemoteObject extends RemoteServer {

    protected UnicastRemoteObject()
        throws java.rmi.RemoteException {...}
    protected UnicastRemoteObject(int port)
        throws java.rmi.RemoteException {...}
    protected UnicastRemoteObject(int port,
                                   RMIClientSocketFactory csf,
                                   RMIServerSocketFactory ssf)
        throws java.rmi.RemoteException {...}

    public Object clone()
        throws java.lang.CloneNotSupportedException {...}
    public static RemoteStub exportObject(java.rmi.Remote obj)
        throws java.rmi.RemoteException {...}
    public static Remote exportObject(java.rmi.Remote obj, int port)
        throws java.rmi.RemoteException {...}
    public static Remote exportObject(Remote obj, int port,
                                       RMIClientSocketFactory csf,
                                       RMIServerSocketFactory ssf)
```

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

5.3.1 *Constructing a New Remote Object*

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

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

5.3.3 *Passing a UnicastRemoteObject in an RMI Call*

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

5.3.4 *Serializing a UnicastRemoteObject*

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

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

5.3.5 *Unexporting a UnicastRemoteObject*

The `unexportObject` method makes the remote object, *obj*, unavailable for incoming calls. If the *force* parameter is true, the object is forcibly unexported even if there are pending calls to the remote object or the remote object still has calls in progress. If the *force* parameter is false, the object is only unexported if there are no pending or in-progress calls to the object. If the object is successfully unexported, the RMI runtime removes the object from its internal

tables. Unexporting the object in this forcible manner may leave clients holding stale remote references to the remote object. This method throws `java.rmi.NoSuchObjectException` if the object was not previously exported to the RMI runtime.

5.3.6 The *clone* method

Objects are only clonable using the Java programming language's default mechanism if they support the `java.lang.Cloneable` interface. The 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 to that remote object. As long as some client holds a remote reference to the remote object, the RMI runtime keeps a local reference to the remote object. Each time the remote object's "reference" set becomes empty (meaning that the number of clients that reference the object becomes zero), the `Unreferenced.unreferenced` method is invoked (if that remote object implements the `Unreferenced` interface). A remote object is *not* required to support the `Unreferenced` interface.

As long as some local reference to the remote object exists, it may be passed in remote calls or returned to clients. The process that receives the reference is added to the reference set for the remote object. When the reference set becomes empty, the remote object's `unreferenced` method will be invoked.

As such, the unreferenced method can be called more than once (each time the set is newly emptied). Remote objects are only collected when no more references, either local references or those held by clients, still exist.

5.5 *The `RMISecurityManager` Class*

See the `RMISecurityManager` API documentation.

5.6 *The `RMIClassLoader` Class*

See the `RMIClassLoader` API documentation.

5.7 *The `LoaderHandler` Interface*

See the `LoaderHandler` API documentation.

5.8 *RMI Socket Factories*

When the RMI runtime implementation needs instances of `java.net.Socket` and `java.net.ServerSocket` for its connections, instead of instantiating objects of those classes directly, it calls the `createSocket` and `createServerSocket` methods on the current `RMISocketFactory` object, returned by the static 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 JDK1.1, this customization was limited to relatively global decisions about socket type, because the only parameters supplied to the factory's methods were `host` and `port` (for `createSocket`) and just `port` (for `createServerSocket`).

In the Java 2 platform, the new interfaces `RMIServerSocketFactory` and `RMIClientSocketFactory` have been introduced to provide more flexible customization of what protocols are used to communicate with remote objects.

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

Remote objects exported with either of the new constructors or `exportObject` methods (with `RMIClientSocketFactory` and `RMISServerSocketFactory` parameters) will be treated differently by the RMI runtime. For the lifetime of such a remote object, the runtime will use the custom `RMISServerSocketFactory` 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 `RMISServerSocketFactory` object that was specified when the remote object was exported.

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

Remote objects exported with the older constructor or method on `UnicastRemoteObject` that do not take custom socket factories as arguments will have `RemoteRef` 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 an integer specifying the port number. This is so that RMI servers that do not use new 1.2 features will interoperate with older RMI clients.

5.8.1 The *RMISocketFactory* Class

The `java.rmi.server.RMISocketFactory` abstract class provides an interface for specifying how the transport should obtain sockets. Note that the class below uses `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*

See the `RMIServerSocketFactory` API documentation.

5.8.3 *The RMIClientSocketFactory Interface*

See the `RMIClientSocketFactory` API documentation.

5.9 *The RMIFailureHandler Interface*

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

```
package java.rmi.server;

public interface RMIFailureHandler {
```



```
        public boolean failure(Exception ex);  
    }
```

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

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

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

5.10 *The LogStream Class*

See the `LogStream` API documentation.

5.11 *Stub and Skeleton Compiler*

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

Please see the following URLs for further information on `rmic`:

- For the Solaris™ operating environment:

<http://java.sun.com/products/j2se/1.4/docs/tooldocs/solaris/rmic.html>

- For the Microsoft Windows platform:

<http://java.sun.com/products/j2se/1.4/docs/tooldocs/win32/rmic.html>

Registry Interfaces



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

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

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

Topics:

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

6.1 The Registry Interface

See the `Registry` API documentation.

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 registry 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 returned 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` method is a specially constructed stub that contains a well-known object identifier. Passing a registry stub from one JVM 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` method creates and exports a registry on the local host on the specified *port*.

The second `createRegistry` method allows more flexibility 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 `RMI ServerSocketFactory`. 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*

See the `RegistryHandler` API documentation.

Remote Object Activation



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 JVM.

7.1.1 Terminology

An *active* object is a remote object that is instantiated and exported in a JVM on some system. A *passive* object is one that is not yet instantiated (or exported) in a JVM, 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 JVM, which may entail loading the class for that object into a JVM 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*, 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 which the class can be loaded, specific data the object may need to bootstrap, etc.), and
- a manager of Java virtual machines, that starts up JVMs (when necessary) and forwards requests for object activation (along with the necessary information) to the correct activation group inside a remote JVM.

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 JVM) is the entity which receives a request to activate an object in the JVM and returns the activated object back to the activator.

The activation protocol 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, specifying the JVM in which it is activated,
- the object's class name,
- a URL path from which to load the object's class code, and

- 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 JVM 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 Java 2 SDK, Standard Edition, v 1.2, 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 which the object’s code can be loaded, and a `MarshaledObject` 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 `MarshaledObject` 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.MarshaledObject data)
        throws ActivationException;

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

    public ActivationDesc(ActivationGroupID groupID,
                          String className,
                          String codebase,
                          java.rmi.MarshaledObject 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 JVM. All objects with the same `ActivationGroupID` are activated in the same JVM. If the current group is inactive an `ActivationException` is thrown. If the *groupID* is null, an `IllegalArgumentException` is thrown.

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 JVM.

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. If 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, which both registers and exports the object, or
- via the first or second `Activatable.exportObject` method, 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 *codebase*, 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 `MarshaledObject`).

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 an 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 JVM'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, an `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

- 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
- 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 JVM, 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 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. 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 JVM, 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` are 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 current activation group for the JVM. 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 `MarshaledObject`

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 JVM for the activation group to establish the group in the new JVM.

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 the corresponding 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 (JVM) 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 JVM (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 `MarshaledObject` 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 JVM. 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:

- a. its objects become active,
- b. its objects become inactive, or
- c. 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 JVM 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 `MarshaledObject` containing object-specific initialization data, and
- returning a `MarshaledObject` 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 JVM.

Such a descriptor contains:

- the group’s class name (a class name of `null` indicates the default `ActivationGroup` implementation),
- 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, which invokes a special constructor that takes two arguments:

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

```
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 application environment overrides (which will override system properties in the group implementation's JVM). The command environment can control the exact command/options used in starting the child JVM, or can be null to accept `rmid`'s default. This constructor will create an `ActivationGroupDesc` with a null group class name, which indicates the system's default `ActivationGroup` implementation.

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 (possibly null). A null group class name indicates the system's default `ActivationGroup` implementation.

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 implementation-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:

- a. its objects become active,
- b. its objects become inactive, or
- c. 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 JVM (as a child process, for example) for each registered activation group and directs activation requests to the appropriate group. It is implementation specific how JVMs 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 `MarshaledObject`)

When created, the default implementation of `ActivationGroup` will set the system properties to the system properties in force when the `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 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 JVM. 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 `MarshallableObject` containing the group's initialization data (obtained from its `ActivationGroupDesc`). If the `ActivationGroupDesc.getClassName` method returns `null`, the system's default group implementation is used. 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 callback 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 JVM. 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 JVM. 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 JVM has become inactive.

7.4.9 The *MarshaledObject* Class

A *MarshaledObject* 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 *MarshaledObject* is serialized and deserialized (when requested) with the same semantics as parameters passed in RMI calls, which means that any remote object in the *MarshaledObject* is represented by a serialized instance of its stub. The object contained by the *MarshaledObject* 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 *MarshaledObject* 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 *MarshaledObject* 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();
}
```

MarshaledObject'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 `MarshaledObject` 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 `MarshaledObject` 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



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) as of the Java 2 SDK, Standard Edition, v1.2.

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 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 programming language 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 objects written in the Java programming language, 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 programming language's threading model. While use of these methods for remote stubs does not break the threading model, these methods do not have the same semantics as they do for local objects written in the Java programming language. Specifically, these methods operate 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 as of the Java 2 SDK, Standard Edition, v1.2. The 1.2 stub protocol does not make use of this interface anymore. As of the Java 2 SDK, Standard Edition, v1.2, 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 `invoke(Remote, Method, Object[], long)` 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 to be used for the *opnum* parameter is a 64-bit (long) integer computed from the first two 32-bit values of the message digest of a particular byte stream using the National Institute of Standards and Technology (NIST) Secure Hash Algorithm (SHA-1). This byte stream contains a string as if it was written using the `java.io.DataOutput.writeUTF` method, consisting of the remote method's name followed by its method descriptor (see section 4.3.3 of *The Java Virtual Machine Specification* (JVMs) for a description of method descriptors).

For example, if a method of a remote interface has the following name and signature:

```
void myRemoteMethod(int count, Object obj, boolean flag)
```

the string containing the remote method's name and descriptor would be the following:

```
myRemoteMethod(ILjava/lang/Object;Z)V
```

The 64-bit hash value is the little-endian composition of an eight byte sequence where the first four bytes are the first 32-bit value of the message digest in big-endian byte order and the last four bytes are the second 32-bit value of the message digest in big-endian byte order. For example, if the first two 32-bit values of the message digest are 0xB0B1B2B3 and 0xB4B5B6B7, then the hash value would be 0xB7B6B5B4B3B2B1B0.

Note – The methods `newCall(RemoteObject, Operation[], int, long)`, `invoke(RemoteCall)`, and `done(RemoteCall)` are deprecated as of the Java 2 SDK, Standard Edition, v1.2. The stubs generated by `rmic` using the 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. The interface *hash* is a 64-bit value used to enforce compatibility between a stub and skeleton using the v1.1 stub protocol. The interface hash is computed from the first two 32-bit values of the message digest of a particular byte stream using SHA-1. This byte stream contains data as if it was written using the `writeInt` and `writeUTF` methods of the interface `java.io.DataOutput`, consisting of the following items:

- (int) stub version number, always 1
- for each remote method, in order of operation number:
 - (UTF-8) remote method name
 - (UTF-8) remote method descriptor (see section 4.3.3 of JVM5)
 - for each declared exception, in lexicographic order of binary name:
 - (UTF-8) the name of the exception class

The interface hash value is composed from the message digest in the same manner as described above for the method hash used in the `invoke` method.

The method `invoke(RemoteCall)` 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` interface was deprecated as of the Java 2 SDK, Standard Edition, v 1.2. 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 as of Java 2 SDK, Standard Edition, v1.2-compatible versions. To generate stubs that are compatible with 1.2 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 method in the Java programming language for a remote object.

Note – The `Operation` interface is deprecated as of the Java 2 SDK, Standard Edition, v1.2. The 1.2 stub protocol no longer uses the old `RemoteRef.invoke` method which takes an `Operation` as one of its arguments. As of the Java 2 SDK, Standard Edition, v1.2, 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 `DGC` Interface
- The `Lease` Class
- The `ObjID` Class
- The `UID` Class
- The `VMID` Class

9.1 The DGC Interface

See the `DGC` API documentation.

9.2 The Lease Class

See the `Lease` API documentation.

9.3 The ObjID Class

See the `ObjID` API documentation.

9.4 *The `UID` Class*

See the `UID` API documentation.

9.5 *The `VMID` Class*

See the `VMID` API documentation.

Topics:

- Overview
- RMI Transport Protocol
- RMI's Use of Object Serialization Protocol
- RMI's Use of HTTP POST Protocol
- Application-Specific Values for RMI
- RMI's Multiplexing Protocol

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 programming language 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 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 written to a Java object output stream that contains 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* represents the method number as assigned by `rmic`, and the *Hash* was the stub/skeleton hash which is the stub's interface hash. As of the Java 2 platform stub protocol (Java 2 platform

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.

<i>Count</i>	short
<i>Exception</i>	java.lang.Exception
<i>Hash</i>	long
<i>Hostname</i>	UTF
<i>Number</i>	int
<i>Object</i>	java.lang.Object
<i>ObjectNumber</i>	long
<i>Operation</i>	int
<i>PortNumber</i>	int
<i>Primitive</i>	byte, int, short, long...
<i>Time</i>	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 uses 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 continuation 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	<i>opcode</i>	operation code (OPEN)
2	<i>ID</i>	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	<i>opcode</i>	operation code (OPEN)
2	<i>ID</i>	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	<i>opcode</i>	operation code (OPEN)
2	<i>ID</i>	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	<i>opcode</i>	operation code (OPEN)
2	<i>ID</i>	connection identifier
4	<i>count</i>	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	<i>opcode</i>	operation code (OPEN)
2	<i>ID</i>	connection identifier
4	<i>count</i>	number of bytes in transmission
<i>count</i>	<i>data</i>	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



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
<code>java.rmi.StubNotFoundException</code>	<ol style="list-style-type: none">1. Class of stub not found.2. Name collision with class of same name as stub causes one of these errors:<ul style="list-style-type: none">• Stub can't be instantiated.• Stub not of correct class.3. Bad URL due to wrong codebase.4. Stub not of correct class.
<code>java.rmi.server.SkeletonNotFoundException</code> <i>note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2</i>	<ol style="list-style-type: none">1. Class of skeleton not found.2. Name collision with class of same name as skeleton causes one of these errors:<ul style="list-style-type: none">• Skeleton can't be instantiated.• Skeleton not of correct class.3. Bad URL due to wrong codebase.4. Skeleton not of correct class.
<code>java.rmi.server.ExportException</code>	The port is in use by another VM.

A.2 Exceptions During RMI Call

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



A.3 Exceptions or Errors During Return

Exception	Context
<code>java.rmi.UnmarshalException</code>	1. Corrupted stream leads to either an I/O or protocol error when: <ul style="list-style-type: none">• Marshaling return header.• Checking return type.• Checking return code.• Unmarshaling return. 2. Return value class not found.
<code>java.rmi.UnexpectedException</code>	An exception not mentioned in the method signature occurred (excluding runtime exceptions). The <code>UnexpectedException</code> exception object contains the underlying exception that was thrown by the server.
<code>java.rmi.ServerError</code>	Any error that occurs while the server is executing a remote method. The <code>ServerError</code> exception object contains the underlying error that was thrown by the server,
<code>java.rmi.ServerException</code>	This exception is thrown as a result of a remote method invocation when a <code>RemoteException</code> is thrown while processing the invocation on the server, either while unmarshalling the arguments or executing the remote method itself. For examples, see Section A.3.1, “Possible Causes of <code>java.rmi.ServerException</code> ”.
<code>java.rmi.ServerRuntimeException</code> <i>note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2</i>	This exception is not thrown by servers running Java 2 SDK, Standard Edition, v1.2-compatible versions. A <code>RuntimeException</code> is propagated to clients intact.

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
<code>java.rmi.server.SkeletonMismatchException</code> <i>note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2</i>	Hash mismatch of stub and skeleton.
<code>java.rmi.UnmarshalException</code>	I/O error unmarshaling call header. I/O error unmarshaling arguments. Invalid method number or method hash.
<code>java.rmi.MarshalException</code>	Protocol error marshaling return.
<code>java.rmi.RemoteException</code>	A <code>RemoteException</code> occurring as a result of a remote invocation by the server.

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
<code>java.rmi.AccessException</code>	Operation disallowed. The registry restricts bind, rebind, and unbind to the same host. The lookup operation can originate from any host.
<code>java.rmi.AlreadyBoundException</code>	Attempt to bind a name that is already bound.
<code>java.rmi.NotBoundException</code>	Attempt to look up a name that is not bound.
<code>java.rmi.UnknownHostException</code>	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
<code>java.rmi.activation.ActivateFailedException</code>	Thrown by RMI runtime when activation fails during a remote call to an activatable object.
<code>java.rmi.activation.ActivationException</code>	General exception class used by the activation interfaces and classes.
<code>java.rmi.activation.UnknownGroupException</code>	Thrown by methods of the activation classes and interfaces when the <code>ActivationGroupID</code> parameter or <code>ActivationGroupID</code> in an <code>ActivationGroupDesc</code> parameter is invalid.
<code>java.rmi.activation.UnknownObjectException</code>	Thrown by methods of the activation classes and interfaces when the <code>ActivationID</code> parameter is invalid.

A.6 Other Exceptions

Exception	Context
<code>java.rmi.RMISecurityException</code> <i>note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2</i>	A security exception that is thrown by the <code>RMISecurityManager</code> .
<code>java.rmi.server.ServerCloneException</code>	Clone failed.
<code>java.rmi.server.ServerNotActiveException</code>	Attempt to get the client host via the <code>RemoteServer.getClientHost</code> method when the remote server is not executing in a remote method.
<code>java.rmi.server.SocketSecurityException</code>	Attempt to export object on an illegal port.



Properties In RMI



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
<code>java.rmi.server.codebase</code>	Indicates the codebase URL of classes originating from the JVM. The codebase property is used to annotate class descriptors of classes <i>originating</i> from a JVM 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.
<code>java.rmi.server.disableHttp</code>	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 <code>false</code> (HTTP usage is enabled).
<code>java.rmi.server.hostname</code>	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 JVM. The property can also be set to an IP address. Not set by default.
<code>java.rmi.dgc.leaseValue</code>	Sets the lease duration that the RMI runtime grants to clients referencing remote objects in the JVM. Defaults to 10 minutes.



Property	Description
<code>java.rmi.server.logCalls</code>	If set to true, server call logging is turned on and prints to stderr. Defaults to false.
<code>java.rmi.server.useCodebaseOnly</code>	If set to true, when RMI loads classes (if not available via CLASSPATH) they are only loaded using the URL specified by the property <code>java.rmi.server.codebase</code> .
<code>java.rmi.server.useLocalHostname</code>	If the <code>java.rmi.server.hostname</code> 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 JVM. Instead, RMI will use the value of the call to the method <code>java.net.InetAddress.getLocalHost</code> .



B.2 Activation Properties

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

Property	Description
<code>java.rmi.activation.port</code>	The port number on which the <code>ActivationSystem</code> is exported. This port number should be specified in a JVM if the activation daemon <code>rmid</code> uses a port other than the default.
<code>java.rmi.activation.activator.class</code>	The class that implements the interface <code>java.rmi.activation.Activator</code> . This property is used internally to locate the resident implementation of the <code>Activator</code> 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 as of Java 2 SDK, Standard Edition, v1.2.

Property	Description
<code>java.rmi.loader.packagePrefix</code> <i>note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2</i>	The package prefix for the class that implements the interface <code>java.rmi.server.LoaderHandler</code> . Defaults to <code>sun.rmi.server</code> .
<code>java.rmi.registry.packagePrefix</code> <i>note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2</i>	The package prefix for the class that implements the interface <code>java.rmi.registry.RegistryHandler</code> . Defaults to <code>sun.rmi.registry</code> .
<code>java.rmi.server.packagePrefix</code> <i>note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2</i>	The server package prefix. Assumes that the implementation of the server reference classes (such as <code>UnicastRef</code> and <code>UnicastServerRef</code>) are located in the package defined by the prefix. Defaults to <code>sun.rmi.server</code> .

