Java™ Remote Method Invocation Specification

Prebeta Draft
Revision 1.1

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

Revision 1.1, November 1, 1996 Draft

Table of Contents

1 Introduction 1

1.1 Overview 1

1.2 System Goals 2

2 Java Distributed Object Model. 5

2.1 Definition of Terms. 5

2.2 The Distributed and Non-Distributed Models Contrasted 6

2.3 RMI Interfaces and Classes 7

2.4 Implementing a Remote Interface 9

2.5 Type Equivalency of Remote Objects with Local Stub . . . 10

2.6 Parameter Passing in Remote Method Invocation 10

2.7 Exception Handling in Remote Method Invocation. 11

2.8 Object Methods Overridden by the RemoteObject Class 11

2.9 The Semantics of Object Methods Declared final 13

2.10 Locating Remote Objects 13

3 System Architecture **15**

3.1 Overview 15

3.2 Architectural Overview 16

3.3 The Stub/Skeleton Layer 18

3.4 The Remote Reference Layer 19

3.5 The Transport Layer 20

3.6 Thread Usage in Remote Method Invocations 21

3.7 Garbage Collection of Remote Objects 21

3.8 Dynamic Class Loading 22

3.9 Security 26

3.10 Configuration Scenarios 27

3.11 RMI Through Firewalls Via Proxies 29

4 Client Interfaces **33**

4.1 The Remote Interface 33

4.2 The RemoteException Class 34

4.3 The Naming Class 34

5 Server Interfaces **37**

5.1 The RemoteObject Class 38

5.2 The RemoteServer Class 38

5.3 The UnicastRemoteObject Class 39

5.4 The Unreferenced Interface 40

5.5 The RMISecurityManager Class 41

5.6 The RMIClassLoader Class 44

5.7 The RMISocketFactory Class 45

5.8 The RMIFailureHandler Interface ... 46
5.9 The LogStream Class ... 47
5.10 Stub and Skeleton Compiler ... 48

6 Registry Interfaces ... 49
6.1 The Registry Interface ... 49
6.2 The LocateRegistry Class ... 51
6.3 The RegistryHandler Interface ... 52

7 Stub/Skeleton Interfaces ... 53
7.1 The RemoteStub Class ... 53
7.2 The RemoteCall Interface ... 54
7.3 The RemoteRef Interface ... 55
7.4 .The ServerRef Interface ... 56
7.5 The Skeleton Interface ... 57
7.6 The Operation Class ... 57

8 Garbage Collector Interfaces ... 59
8.1 The Interface DGC ... 59
8.2 The Lease Class ... 61
8.3 The ObjID Class ... 61
8.4 The UID Class ... 62
8.5 The VMID Class ... 63

A Exceptions In RMI ... 65
A.1 Exceptions During Remote Object Export ... 66
A.2 Exceptions During RMI Call ... 67
A.3 Exceptions or Errors During Return ... 67

A.4 Other 68

Introduction 1

Topics:

- Overview
- System Goals

1.1 Overview

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

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

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

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

1.2 System Goals

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

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

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

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

- Several invocation mechanisms, for example simple invocation to a single object or invocation to an object replicated at multiple locations. The system should also be extensible to other invocation paradigms.

- Various reference semantics for remote objects, for example live (non-persistent) references, persistent references, and lazy activation.
- The safe Java environment provided by security managers and class loaders.
- Distributed garbage collection of active objects.
- Capability of supporting multiple transports.

The first two chapters in this specification describe the distributed object model for the Java language and the system architecture. The remaining chapters describe the client and server visible interfaces that comprise the Java RMI to be released with the JDK 1.1.

Java Distributed Object Model 2

Topics:

- Definition of Terms
- The Distributed and Non-Distributed Models Contrasted
- RMI Interfaces and Classes
- Implementing a Remote Interface
- Type Equivalency of Remote Objects with Local Stub
- Parameter Passing in Remote Method Invocation
- Exception Handling in Remote Method Invocation
- Object Methods Overridden by the RemoteObject Class
- The Semantics of Object Methods Declared final
- Locating Remote Objects

2.1 *Definition of Terms*

In the Java distributed object model, a *remote object* is one whose methods can be invoked from another Java Virtual Machine, potentially on a different host. An object of this type is described by one or more *remote interfaces*, which are Java interfaces that declare the methods of the remote object.

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

2.2 The Distributed and Non-Distributed Models Contrasted

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

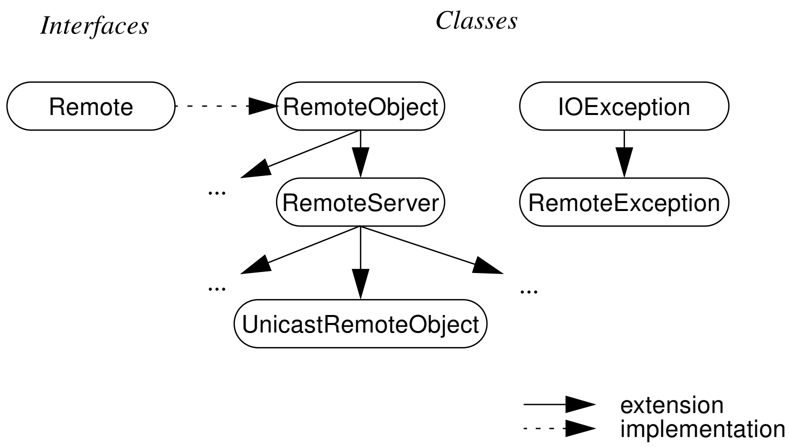
- A reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote).
- A remote object can be cast to any of the set of remote interfaces supported by the implementation using the built-in Java syntax for casting.
- The built-in Java `instanceof`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

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

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

2.3 RMI Interfaces and Classes

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



2.3.1 The Remote Interface

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

```
public interface Remote {}
```

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

```
public interface BankAccount
       extends Remote
{
    public void deposit (float amount)
        throws java.rmi.RemoteException;
    public void withdraw (float amount)
        throws OverdrawnException, java.rmi.RemoteException;
    public float balance()
        throws java.rmi.RemoteException;
}
```

The methods in a remote interface must be defined as follows:

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

2.3.2 The RemoteException Class

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

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

2.3.3 The RemoteObject Class and its Subclasses

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

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

2.4 Implementing a Remote Interface

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

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

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

```
package my_package;

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

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

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

2.5 Type Equivalency of Remote Objects with Local Stub

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

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

Stubs are generated using the rmic compiler.

2.6 Parameter Passing in Remote Method Invocation

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

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

2.6.1 Passing non-remote objects

A non-remote object passed as a parameter of a remote method invocation or returned as a result of a remote method invocation is passed by *copy*.

That is, when a non-remote object appears in a remote method invocation, the content of the non-remote object is copied before invoking the call on the remote object. By default, only the non-static and non-transient fields are copied.

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

2.6.2 Passing remote objects

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

2.7 Exception Handling in Remote Method Invocation

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

2.8 Object Methods Overridden by the RemoteObject Class

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

2.8.1 equals and hashCode

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

- The `java.rmi.server.RemoteObject` class's implementation of the `equals` method determines whether two object references are equal, not whether the contents of the two objects are equal. This is because determining equality based on content requires a remote method invocation, and the signature of `equals` does not allow a remote exception to be thrown.
- The `java.rmi.server.RemoteObject` class's implementation of the `hashCode` method returns the same value for all remote references that refer to the same underlying remote object (because references to the same object are considered equal).

2.8.2 toString

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

2.8.3 clone

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

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

2.8.4 finalize

Remote object implementations (i.e subclasses of `RemoteObject`) may use `finalize` to perform their own cleanup as necessary, for example to deactivate an object server.

2.9 The Semantics of Object Methods Declared final

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

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

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

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

2.10 Locating Remote Objects

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

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

Here's an example, (without exception handling) of how to bind and lookup remote objects:

```
BankAccount acct = new BankAcctImpl();
String url = "rmi://java.Sun.COM/account";
// bind url to remote object
java.rmi.Naming.bind(url, acct);
    ...
// lookup account
acct = (BankAccount)java.rmi.Naming.lookup(url);
```

System Architecture 3

Topics:

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

3.1 Overview

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

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

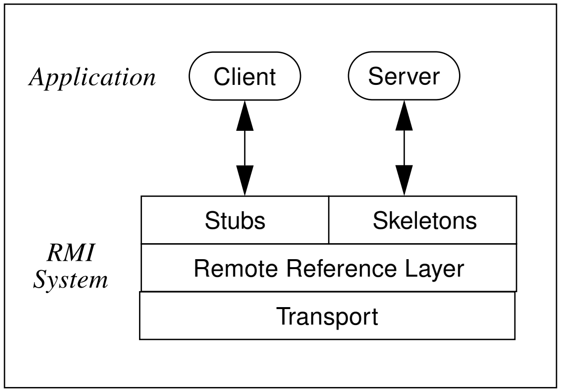
Another technique, called *dynamic stub loading*, is used to support client-side stubs which implement the same set of remote interfaces as a remote object itself. This technique, used when a stub of the exact type is not already available to the client, allows a client to use the Java language's built-in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

3.2 Architectural Overview

The RMI system consists of three layers:

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

The application layer sits on top of the RMI system. The relationship between the layers is shown in the following figure.



A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

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

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

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

The *transport* is responsible for connection set-up, connection management, and keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport's address space.

In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to be done before handing off the request to the server-side skeleton. The skeleton for a remote object makes an up-call to the remote object implementation which carries out the actual method call.

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

3.3 The Stub/Skeleton Layer

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

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

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

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

- Unmarshaling arguments from the marshal stream.
- Making the up-call to the actual remote object implementation.

- Marshaling the return value of the call or an exception (if one occurred) onto the marshal stream.

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

3.4 The Remote Reference Layer

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

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

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

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

In a corresponding manner, the server-side component implements the specific remote reference semantics prior to delivering a remote method invocation to the skeleton. This component, for example, could handle ensuring atomic multicast delivery by communicating with other servers in the replica group.

The remote reference layer transmits data to the transport layer via the abstraction of a stream-oriented *connection*. The transport takes care of the implementation details of connections. Although connections present a streams-based interface, a connectionless transport may be implemented beneath the abstraction.

3.5 The Transport Layer

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

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

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

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

- An *endpoint* is the abstraction used to denote an address space or Java virtual machine. In the implementation, an endpoint can be mapped to its transport. That is, given an endpoint, a specific transport instance can be obtained.
- A *channel* is the abstraction for a conduit between two address spaces. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel.
- A *connection* is the abstraction for transferring data (performing input/output).

- The *transport* abstraction manages channels. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces, the local address space and a remote address space. Given an endpoint to a remote address space, a transport sets up a channel to that address space. The transport abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system.

A transport defines what the concrete representation of an endpoint is, so multiple transport implementations may exist. The design and implementation also supports multiple transports per address space, so both TCP and UDP can be supported in the same virtual machine.

3.6 Thread Usage in Remote Method Invocations

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

3.7 Garbage Collection of Remote Objects

In a distributed system, just as in the local system, it is desirable to automatically delete those remote objects that are no longer referenced by any client. This frees the programmer from needing to keep track of the remote objects clients so that it can terminate appropriately. RMI uses a reference counting garbage collection algorithm similar to Modula-3's Network Objects. (see "Network Objects" by Birrell, Nelson, and Owicki, *Digital Equipment Corporation Systems Research Center Technical Report 11*5, 1994)

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

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

As long as a local reference to a remote object exists, it cannot be garbage collected and it may 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 may be called more than once. Remote objects are only collected when no more references, either local or remote, still exist.

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

3.8 Dynamic Class Loading

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

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

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

This section describes

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

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

3.8.1 How a Class Loader is Chosen

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

- The AppletClassLoader is used to download a Java applet over the net from the location specified by the codebase attribute on the web page that contains the <applet> tag. All classes used directly in the applet are subsequently loaded by the AppletClassLoader.
- The default class loader is used to load a Java application (either a client or server) from the local CLASSPATH. All classes used directly in the application are subsequently loaded by the default class loader from the local CLASSPATH.

- The RMIClassLoader is used to load those classes not directly used by the client or server application: the stubs of remote objects, remote interfaces, and the extended classes of arguments and return values to RMI calls. The RMIClassLoader looks for these classes in the following locations, in the order listed:

    a. The local CLASSPATH. Classes are always loaded locally if they exist locally.

    b. For remote object references passed as parameters or return values, the URL encoded in the marshal stream that contains the serialized object.

    c. The URL specified by the local `java.rmi.server.codebase` property.

For case b, the URL of a remote object is passed in the stream only if the `RMIClassLoader` on the virtual machine where the remote object implementation resides loaded the class from the location specified by the `java.rmi.server.codebase` property (or the `AppletClassLoader` loaded the class from the applet's codebase). If the class was loaded from CLASSPATH, the URL is not sent even if the `java.rmi.server.codebase` property is set.

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

3.8.2 Bootstrapping the Client

For the RMI runtime to be able to download *all* the classes and interfaces needed by a client application, a bootstrapping client program is required which forces the use of the `RMIClassLoader` instead of the default class loader. The bootstrapping program needs to:

- Create an instance of the `RMISecurityManager` or user-defined security manager.
- Use the `RMIClassLoader` to load the class file for the client. The class name cannot be mentioned explicitly in the code, but must instead be a string or a command line argument. Otherwise, the default class loader will try to load the client class file from the local CLASSPATH.

- Use the `newInstance` method to create an instance of the client and cast it to Runnable (the client must implement the `java.lang.Runnable` interface
- Start the client by calling the run method (inherited from Runnable).

For example:

```
public class LoadClient
{
    public static void main(String args[])
    {
        System.setSecurityManager(new RMISecurityManager());

        try {
            RMIClassLoader loader = RMIClassLoader.getLocalLoader();
            Class c = loader.loadClass(args[0]);
            Runnable client = c.newInstance();
            client.run();
        } catch (ClassNotFoundException e) {
            System.out.println("Exception: " + e.getMessage());
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

This example assumes that the bootstrapping program is executed like this:

```
java LoadClient -Djava.rmi.server.codebase=http://host/rmiclasses/
MyClient
```

where `MyClient` is the name of the client class read in as `args[0]`.

Once the client is started and has control, all classes needed by the client will also be loaded by the RMIClassLoader from `java.rmi.server.codebase`. This bootstrapping technique is exactly the same technique Java uses to force the AppletClassLoader to download the same classes used in an applet.

Without this bootstrapping technique, all the code needed by the client must be available through the local CLASSPATH on the client, and the only Java code that can be loaded by the RMIClassLoader over the net is class files that are not used directly the client program. These classes are remote interfaces, stubs, and the extended classes of arguments and return values to remote method invocations.

3.9 Security

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

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

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

Applications must either define their own security manager or use the restrictive `RMISecurityManager`. If no security manager is in place, an application cannot load classes from network sources.

A client or server program is usually implemented by classes loaded from the local system and therefore is not subject to the restrictions of the security manager. If however, the client program itself is downloaded from the network using the technique described in Section 3.8.2, "Bootstrapping the Client", then the client program is subject to the restrictions of the security manager.

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

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

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

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

3.10 Configuration Scenarios

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

3.10.1 Servers

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

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

3.10.2 Applets

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

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

3.10.3 Applications

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

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

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

To enable downloading from a network source, the host where the classes reside must be configured with the `java.rmi.server.codebase` property. This allows the RMI system to embed the URL of a class in the serialized form of the class.

Note – Classes on host A which are visible to other hosts via host A's `java.rmi.server.codebase` property cannot also be available through host A's CLASSPATH. If they are, the RMI system will not include the URL of the class when it is serialized. The easiest way to safeguard against this is to put all classes on an HTTP server and have all the applications (clients, servers, and peers) load them from there.

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

3.11 RMI Through Firewalls Via Proxies

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

3.11.1 How an RMI Call is Packaged within the HTTP Protocol

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

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

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

3.11.2 The Default Socket Factory

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

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

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

3.11.3 Configuring the Client

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

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

3.11.4 *Configuring the Server*

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 VM 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, to start the RMI server class `ServerImpl` on the machine chatsubo.javasoft.com:

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

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.

---

2. The following configuration is necessary only if the server will not support RMI clients behind firewalls that can forward to arbitrary ports:

   a. An HTTP server is listening on port 80.

   b. A CGI script is located at the aliased URL path `/cgi-bin/java-rmi`. This script:

      - Invokes the local Java interpreter to execute a class internal to the transport layer which forwards the request to the appropriate RMI server port.
      - Defines properties in the Java virtual machine with the same names and values as the CGI 1.0 defined environment variables.

   An example script is supplied in the RMI distribution for the Solaris and Win32 operating systems. Note that the script must specify the complete path to the java interpreter on the server machine.

3.11.5 Performance Issues and Limitations

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

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

Client Interfaces 4

When writing an Applet or an application that uses remote objects, the programmer may need to be aware of the RMI system's client visible interfaces.

Topics:

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

4.1 The Remote Interface

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

The `java.rmi.Remote` interface serves to identify all remote objects, any object that is a remote object must directly or indirectly implement this interface. All remote interfaces must be declared `public`.

4.2 The RemoteException Class

All remote exceptions are subclasses of `java.rmi.RemoteException`. This allows interfaces to handle all types of remote exceptions and to distinguish local exceptions and exceptions specific to the method from exceptions thrown by the underlying distributed object mechanisms.

```
package java.rmi;
public class RemoteException
    extends java.io.IOException
{
    // The actual exception or error that occured.
    public Throwable detail;

    // Create a remote exception.
    public RemoteException()

    // Create a remote exception with the specified string.
    public RemoteException(String s)

    // Create remote exception with specified string and exception.
    public RemoteException(String s, Throwable ex)

    // Produce message, including message from any nested exception.
    public String getMessage()
}
```

4.3 The Naming Class

The `java.rmi.Naming` class allows remote objects to be retrieved and defined using the familiar Uniform Resource Locator (URL) syntax. The URL consists of protocol, host, port and name fields. The `Registry` service on the specified host and port is used to perform the specified operation. The protocol should be specified as `rmi`, as in `rmi://java.sun.com:2001/root`.

```
package java.rmi;
public final class Naming {
    public static Remote lookup(String url)
        throws NotBoundException, java.net.MalformedURLException,
        UnknownHostException, RemoteException;

    public static void bind(String url, Remote obj)
        throws AlreadyBoundException,
        java.net.MalformedURLException, UnknownHostException,
        RemoteException;

    public static void rebind(String url, Remote obj)
        throws RemoteException, java.net.MalformedURLException,
        UnknownHostException;

    public static void unbind(String url)
        throws RemoteException, NotBoundException,
        java.net.MalformedURLException, UnknownHostException;

    public static String[] list(String url)
        throws RemoteException, java.net.MalformedURLException,
        UnknownHostException;
}
```

The `lookup` method returns the remote object associated with the file portion of the name; so in the preceding example it would return the object named `root`. The `NotBoundException` is thrown if the name has not been bound to an object.

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

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

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

The list `method` returns an array of `Strings` containing a snapshot of the URLs bound in the Registry. The file part of the URL is ignored.

Note – The `java.rmi.AccessException` may also be thrown as a result of any of these methods.

Server Interfaces 5

When implementing a server, the client interfaces are available and extended with those that allow the definition, creation and export of remote objects.

Topics:

- The RemoteObject Class
- The RemoteServer Class
- The UnicastRemoteObject Class
- The Unreferenced Interface
- The RMISecurityManager Class
- The RMIClassLoader Class
- The RMISocketFactory Class
- The RMIFailureHandler Interface
- The LogStream Class
- Stub and Skeleton Compiler

5.1 The RemoteObject Class

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

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

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

```
package java.rmi.server;
public abstract class RemoteObject
    implements java.rmi.Remote, java.io.Serializable
{
    public int hashCode();
    public boolean equals(Object obj);
    public String toString();
}
```

5.2 The RemoteServer Class

The `java.rmi.server.RemoteServer` class is the common superclass to all server implementations and provides the framework to support a wide range of remote reference semantics. At present the only subclass supported is `UnicastRemoteObject`.

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

    public static String getClientHost()
        throws ServerNotActiveException;

    public static void setLog(java.io.OutputStream out);

    public static java.io.PrintStream getLog();
}
```

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

5.3 The UnicastRemoteObject Class

The `java.rmi.server.UnicastRemoteObject` class provides support for point-to-point active object references using TCP-based streams. The class implements a remote server object with the following characteristics:

- References are valid only for at most the life of the process that creates the remote object.
- A TCP connection based transport is used.
- 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;

    public Object clone()
        throws java.lang.CloneNotSupportedException;

    public static void exportObject(java.rmi.Remote obj)
        throws java.rmi.RemoteException;
}
```

5.3.1 Constructing a new Remote Object

In a Java virtual machine running as a server, remote objects defined by the developer may be created by the server application. When a remote object class extends `UnicastRemoteObject`, the constructor creates and exports a remote object. The constructor is invoked from the corresponding constructor of the remote object class. The default constructor creates a new unicast remote object using an anonymous port.

The clone method is used to create a unicast remote object with initially the same contents, but is exported to accept remote calls and is distinct from the original object.

5.3.2 Exporting an implementation not extended from RemoteObject

The `exportObject` method is used to export a unicast remote object that is not implemented by extending `UnicastRemoteObject` class. The `exportObject` method is called with the object to be exported on an anonymous port. The object must be exported prior to the first time it is passed in an RMI call as either a parameter or return value; otherwise a `java.rmi.server.StubNotFoundException` is thrown when a remote call is attempted.

Once exported, the object can be passed as an argument in an RMI call or returned as the result of an RMI call. When a remote object is passed, during marshaling a lookup is performed to find the matching remote stub for the remote object implementation and that stub is passed or returned instead.

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 remote references to it. The distributed garbage collection mechanism maintains a set of remote references to each remote object. As long as some client holds a remote reference the RMI runtime keeps a local reference to the remote object. When the set becomes empty the `Unreferenced.unreferenced` method is invoked. No action is required by the implementation and it is not required to support `Unreferenced`.

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 reference. When those new references no longer exist `Unreferenced` will be invoked. As such, the `Unreferenced` method may be called more than once, each time the set is newly emptied. Remote objects are only collected when no more references, either local references or those held by clients, still exist.

5.5 The RMISecurityManager Class

```
package java.rmi;

public class RMISecurityManager extends java.lang.SecurityManager {

    // Constructor
    public RMISecurityManager();

    // The host stub came from determines what stub can do
    public Object getSecurityContext();

    //Disallow creating class loaders or execute ClassLoader methods
    public synchronized void checkCreateClassLoader()
        throws RMISecurityException;

    // Disallow thread manipulation
    public synchronized void checkAccess(Thread t)
        throws RMISecurityException;

    // Disallow thread group manipulation.
    public synchronized void checkAccess(ThreadGroup g)
        throws RMISecurityException;

    // Disallow exiting the VM
    public synchronized void checkExit(int status)
        throws RMISecurityException;

    // Disallow forking of processes
    public synchronized void checkExec(String cmd)
        throws RMISecurityException;

    // Disallow linking dynamic libraries
    public synchronized void checkLink(String lib)
        throws RMISecurityException;

    // Disallow accessing of all properties except those labeled OK
    public synchronized void checkPropertiesAccess()
        throws RMISecurityException;

    // Access system property key only if key.stub is set to true
    public synchronized void checkPropertyAccess(String key)
        throws RMISecurityException;

    // Check if a stub can read a particular file.
```

```
public synchronized void checkRead(String file)
    throws RMISecurityException;

// Check if a stub can read a particular file.
public synchronized void checkRead(String file, URL base)
    throws RMISecurityException;

// No file reads are valid from a stub
public void checkRead(String file, Object context)
    throws RMISecurityException;

// Check if a Stub can write a particular file.
public synchronized void checkWrite(String file)
    throws RMISecurityException;

// Check if the specified system dependent file can be deleted.
public void checkDelete(String file)
    throws RMISecurityException;

// Disllow opening file descriptor for reading unless via socket
public synchronized void checkRead(FileDescriptor fd)
    throws RMISecurityException;

// Disallow opening file descriptor for writing unless via socket
public synchronized void checkWrite(FileDescriptor fd)
    throws RMISecurityException;

// Disallow listening on any port.
public synchronized void checkListen(int port)
    throws RMISecurityException;

// Disallow accepting connections on any port.
public synchronized void checkAccept(String host, int port)
    throws RMISecurityException;

// Disallow making connections on any port.
public synchronized void checkConnect(String host, int port)
    throws RMISecurityException;

// Disallow making connections on any port.
public void checkConnect(String host, int port, Object context)
    throws RMISecurityException;

// Disallow making connections on any port.
public synchronized void checkConnect(String fromHost,
    String toHost)
```

```
        throws RMISecurityException;

    // Allow caller to create top-level windows.
    // Allow stubs to create windows with warnings.
    public synchronized boolean checkTopLevelWindow(Object window)
        throws RMISecurityException;

    // Check if a stub can access a package.
    public synchronized void checkPackageAccess(String pkg)
        throws RMISecurityException;

    // Check if a stub can define classes in a package.
    public synchronized void checkPackageDefinition(String pkg)
        throws RMISecurityException;

    // Check if a stub can set a networking-related object factory.
    public synchronized void checkSetFactory()
        throws RMISecurityException;

    // Disallow printing from stubs.
    public void checkPrintJobAccess()
        throws RMISecurityException;

    // Checks to see if client code can access class members.
    // Allow access to all public information. Allow non-stubs to
    // access default, package, and private declarations and data).
    public void checkMemberAccess(Class clazz, int which)
        throws RMISecurityException;
}
```

The `RMISecurityManager` can be used when the application does not require specialized security functions but does need the protection it provides. This simple security manger disables all functions except class definition and access so that other classes for remote objects, their arguments and returns can be loaded as needed.

If no security manager has been set, stub loading is disabled. This insures that some security manager must be responsible for the actions of loaded stubs and classes as part of any remote method invocation. A security manager is set using `System.setSecurityManager`.

5.6 The RMIClassLoader Class

The RMI runtime uses its own class loader to load stubs, skeletons, and other classes needed by the stubs and skeletons. These classes and the way they are used support the safety properties of the Java RMI runtime. This class loader always loads locally available classes first. Only if a security manager is in force will stubs be loaded from either the local machine or from a network source.

The class loader keeps a cache of loaders for individual Uniform Resource Locators (URLs) and the classes that have been loaded from them. When a stub or skeleton has been loaded, any class references that occur as parameters or returns will be loaded (from their originating codebase host) and are subject to the same security restrictions.

Server processes must declare to the RMI runtime the location of the classes (stubs and parameters/returns) that will be available to its clients. The `java.rmi.server.codebase` property should be a URL from which stub classes and classes used by stubs will be loaded, using the normal protocols, for example http, ftp, etc...

The `java.rmi.server.RMIClassLoader` is a utility class that can be used by applications to load classes via a URL.

```
package java.rmi.server;

public class RMIClassLoader {

    public static Class loadClass(String name)
    throws MalformedURLException, ClassNotFoundException

    public static synchronized Class loadClass(URL codebase,
        String name) throws MalformedURLException,
        ClassNotFoundException
}
```

The first `loadClass` method loads the specified class *name* via the URL defined by the `java.rmi.server.codebase` property. The class is loaded, defined, and returned.

The second form of the loadClass method loads the specified class name via the URL parameter *codebase*.

5.7 The RMISocketFactory Class

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

```
package java.rmi.server;
public abstract class RMISocketFactory {

    public abstract java.net.Socket createSocket(String h,int p)
        throws IOException;

    public abstract java.net.ServerSocket createServerSocket(int p)
        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 gets 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.

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, and 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, and 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 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.

```
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.Socket` or `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, creation is not attempted.

5.9 The LogStream Class

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

```
package java.rmi.server;

public class LogStream extends PrintStream {

    public static LogStream log(String name);

    public static void showThreadName(boolean doIt);

    public static synchronized PrintStream getDefaultStream();

    public static synchronized void setDefaultStream(
        PrintStream newDefault);

    public synchronized OutputStream getOutputStream();

    public synchronized void setOutputStream(OutputStream out);

    public void write(int b);

    public void write(byte b[], int off, int len);

    public String toString();

    public static int parseLevel(String s);

    // constants for logging levels
    public static final int SILENT  = 0;
    public static final int BRIEF   = 10;
    public static final int VERBOSE = 20;
}
```

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

The method `showThreadName` sets whether thread names are part of the message label.

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

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

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

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

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

The method `toString` returns log name as string representation

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

5.10 Stub and Skeleton Compiler

The `rmic` stub and skeleton compiler is used to compile the appropriate stubs and skeletons for a specific remote object implementation. The compiler is invoked with the package qualified class name of the remote object class. The class must previously have been compiled successfully.

- The location of the imported classes may be specified either with the CLASSPATH environment variable or with the -classpath argument.
- The compiled class files are placed in the current directory unless the -d argument is specified following the same mechanism for writing class files as javac.
- The -keepgenerated argument retains the generated java source files for the stubs and skeletons.
- The -show option displays a graphical user interface for the program.

Registry Interfaces 6

The RMI system uses the `java.rmi.registry.Registry` interface and the `java.rmi.registry.LocateRegistry` class to provide a well known bootstrap service for retrieving and registering objects by simple names. Any server process can support its own registry or a single registry can be used for a host.

A Registry is a remote object that maps names to remote objects. A Registry can be used in a virtual machine with other server classes or stand-alone.

The methods of `LocateRegistry` are used to get a Registry operating on a particular host or host and port.

Topics:

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

6.1 The Registry Interface

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

```
package java.rmi.registry;

public interface Registry extends java.rmi.Remote
{
    public java.rmi.Remote lookup(String name)
        throws java.rmi.RemoteException,
        java.rmi.NotBoundException, java.rmi.AccessException;

    public void bind(String name, java.rmi.Remote obj)
        throws java.rmi.RemoteException,
        java.rmi.AlreadyBoundException, java.rmi.AccessException;

    public void rebind(String name, java.rmi.Remote obj)
        throws java.rmi.RemoteException, java.rmi.AccessException;

    public void unbind(String name)
        throws java.rmi.RemoteException,
        java.rmi.NotBoundException, java.rmi.AccessException;

    public String[] list()
        throws java.rmi.RemoteException, java.rmi.AccessException;
}
```

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

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

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

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

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

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

6.2 The LocateRegistry Class

The class `java.rmi.registry.LocateRegistry` contains static methods that retrieve a registry on the current host, current host at specified port, a specified host or at a particular port on a specified host.

```
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,
        java.rmi.UnknownHostException;

    public static Registry getRegistry(String host, int port)
        throws java.rmi.RemoteException,
        java.rmi.UnknownHostException;

    public static Registry createRegistry(int port)
        throws java.rmi.RemoteException;
}
```

The `createRegistry` method creates and exports a registry on the local host on the specified *port*. The registry implements a simple flat naming syntax that binds the name of a remote object (a string) to a remote object reference. The name and remote object bindings are not remembered across server restarts.

Note – Starting a registry with the `createRegistry` method keeps the server process alive.

6.3 The RegistryHandler Interface

The interface `RegistryHandler` is used to interface to the private implementation.

```
package java.rmi.registry;

public interface RegistryHandler {

    Registry registryStub(String host, int port)
        throws java.rmi.RemoteException,
        java.rmi.UnknownHostException;

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

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

The method `registryImpl` constructs and exports a Registry on the specified port. The port must be non-zero.

Stub/Skeleton Interfaces 7

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 Skeleton Interface
- The Operation Class

7.1 The RemoteStub Class

The `java.rmi.server.RemoteStub` class is the common superclass to all client stubs. 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 {}
```

7.2 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.

```
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 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 a 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.

7.3 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 {

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

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

7.4 .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,
        RemoteServer server, 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 *server* is the remote server object for the implementation (may be the same as *obj*), and the parameter *data* contains information necessary to export the object (e.g. 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 hostname of the client invoking the call is returned. If a remote method call is not currently being service, then ServerNotActiveException is called.

7.5 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.

```
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.

7.6 The Operation Class

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

```
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 8

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

Topics:

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

8.1 The Interface DGC

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

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

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

public interface DGC extends java.rmi.Remote {

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

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

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

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

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

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

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

8.2 The Lease Class

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

```
package java.rmi.dgc;

public final class Lease implements java.io.Serializable {

    public Lease(VMID id, long duration);

    public VMID getVMID();

    public long getValue();
}
```

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

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

The `getValue` method returns the lease duration (either requested or granted).

8.3 The ObjID Class

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

```
package java.rmi.server;

public final class ObjID implements java.io.Serializable {
```

```
    public ObjID ();

    public ObjID (int num);

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

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

    public int hashCode()

    public boolean equals(Object obj)

    public String toString()
}
```

The first form of the `ObjID` constructor generates a unique object identifier. The second constructor generates *well-known* object identifiers (e.g. used by the registry and the distributed garbage collector) and takes as an argument a well-known object number. A well-known object ID generated via this second constructor will not clash with any object IDs generated via the default constructor.

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

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

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

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

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

8.4 The UID Class

The class `UID` is an abstraction for creating identifiers that are unique with respect to the host on which it is generated. A `UID` is contained in an `ObjID` as an address space identifier.

```
package java.rmi.server;

public final class UID implements java.io.Serializable {
```

```
    public UID();

    public UID(short num);

    public int hashCode();

    public boolean equals(Object obj);

    public String toString();

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

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

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

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

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

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

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

8.5 The VMID Class

The class `VMID` provides a universally unique identifier for a client VM. A `VMID` contains a `UID` and a host address.

```
package java.rmi.dgc;

public final class VMID implements java.io.Serializable {

    public VMID() throws VMIDNotSupportedException;
```

```
    public boolean equals(Object obj);

    public String toString();
}
```

The VMID default constructor creates a globally unique identifier for the VM. This identifier contains the host address of the machine on which it was created. Due to security restrictions, obtaining the local host address is not always possible. If this restriction is in force, then the constructor will throw `VMIDNotSupportedException`.

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

Exceptions In RMI A

Topics:

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

A.1 Exceptions During Remote Object Export

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

Exception	**Context**
java.rmi.StubNotFoundException	1.) Class of stub not found. 2.) Name collision with class of same name as stub leads to: a.) Stub can't be instantiated. b.) Stub not of correct class. 3.) Bad URL due to wrong codebase. 4.) Stub not of correct class.
java.rmi.server.SkeletonNotFoundException	1.) Class of skeleton not found. 2.) Name collision with class of same name as skeleton leads to: a.) Skeleton can't be instantiated. b.) Skeleton not of correct class. 3.) Bad URL due to wrong codebase. 4.) Skeleton not of correct class.
java.rmi.server.ExportException	The port is in use by another VM.

A.2 Exceptions During RMI Call

Exception	Context
java.rmi.UnknownHostException	Unknown host.
java.rmi.ConnectException	Connection refused to host.
java.rmi.ConnectIOException	I/O error creating connection.
java.rmi.MarshalException	I/O error marshaling transport header, marshaling call header or marshaling arguments.
java.rmi.StubNotFoundException	Remote object not exported.

A.3 Exceptions or Errors During Return

Exception	Context
java.rmi.UnmarshalException	1.) Corrupted stream leads to I/O or protocol error: a.) Unmarshaling return header. b.) Checking return type. c.) Checking return code. d.) Unmarshaling return. 2.) Return value class not found.
java.rmi.UnexpectedException	An exception not in the method signature occurred, including runtime exceptions on the client. This exception object contains the underlying exception.
java.rmi.ServerRuntimeException	Any runtime exception that occurs while the server is executing a method, even if the exception is in the method signature. This exception object contains the underlying exception.
java.rmi.ServerException	Any remote exception that occurs while the server is executing a remote method. See Section A.3.1 for examples.
java.rmi.server.ServerError	Any error that occurs while the server is executing a remote method.

A.3.1 Possible Causes of java.rmi.ServerException

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

Exception	Context
java.rmi.server.SkeletonMismatchException	Hash mismatch of stub and skeleton.
java.rmi.UnmarshalException	I/O error unmarshaling call header. I/O error unmarshaling arguments.
java.rmi.MarshalException	Protocol error marshaling return.
java.rmi.RemoteException	Method number out of range due to corrupted stream.

A.4 Other

Exception	Context
java.rmi.server.ServerCloneException	Clone failed