With *Java RMI*, you'll learn tips and tricks for making your RMI code excel. This book provides strategies for working with serialization, threading, the RMI registry, sockets and socket factories, activation, dynamic class downloading, HTTP tunneling, distributed garbage collection, JNDI, and CORBA. In short, a treasure trove of valuable RMI knowledge packed into one book.

---

**I: Designing and Building: The Basics of RMI Applications**

1. Streams
   1.1 The Core Classes
   1.2 Viewing a File
   1.3 Layering Streams
   1.4 Readers and Writers

2. Sockets
   2.1 Internet Definitions
   2.2 Sockets
   2.3 ServerSockets
   2.4 Customizing Socket Behavior
   2.5 Special-Purpose Sockets
   2.6 Using SSL

3. A Socket-Based Printer Server
10.7 The Externalizable Interface

11. Threads
   11.1 More Than One Client
   11.2 Basic Terminology
   11.3 Threading Concepts
   11.4 Support for Threads in Java
   11.5 Deadlock
   11.6 Threading and RMI

12. Implementing Threading
   12.1 The Basic Task
   12.2 Guidelines for Threading
   12.3 Pools: An Extended Example
   12.4 Some Final Words on Threading

13. Testing a Distributed Application
   13.1 Testing the Bank Application

14. The RMI Registry
   14.1 Why Use a Naming Service?
   14.2 The RMI Registry
   14.3 The RMI Registry Is an RMI Server
   14.4 Examining the Registry
   14.5 Limitations of the RMI Registry
   14.6 Security Issues

15. Naming Services
   15.1 Basic Design, Terminology, and Requirements
   15.2 Requirements for Our Naming Service
   15.3 Federation and Threading
   15.4 The Context Interface
   15.5 The Value Objects
   15.6 ContextImpl
   15.7 Switching Between Naming Services
   15.8 The Java Naming and Directory Interface (JNDI)

16. The RMI Runtime
   16.1 Reviewing the Mechanics of a Remote Method Call
   16.2 Distributed Garbage Collection
   16.3 RMI's Logging Facilities
   16.4 Other JVM Parameters

17. Factories and the Activation Framework
   17.1 Resource Management
   17.2 Factories
   17.3 Implementing a Generic Factory
   17.4 A Better Factory
   17.5 Persistence and the Server Lifecycle
   17.6 Activation
   17.7 A Final Word About Factories

III: Advanced Topics
18. Using Custom Sockets
   18.1 Custom Socket Factories
   18.2 Incorporating a Custom Socket into an Application

19. Dynamic Classloading
   19.1 Deploying Can Be Difficult
   19.2 Classloaders
   19.3 How Dynamic Classloading Works
   19.4 The Class Server
   19.5 Using Dynamic Classloading in an Application

20. Security Policies
   20.1 A Different Kind of Security Problem
   20.2 Permissions
   20.3 Security Managers
   20.4 Setting Up a Security Policy

21. Multithreaded Clients
   21.1 Different Types of Remote Methods
   21.2 Handling Printer-Type Methods
   21.3 Handling Report-Type Methods
   21.4 Generalizing from These Examples

22. HTTP Tunneling
   22.1 Firewalls
   22.2 CGI and Dynamic Content
   22.3 HTTP Tunneling
   22.4 A Servlet Implementation of HTTP Tunneling
   22.5 Modifying the Tunneling Mechanism
   22.6 The Bank via HTTP Tunneling
   22.7 Drawbacks of HTTP Tunneling
   22.8 Disabling HTTP Tunneling

23. RMI, CORBA, and RMI/IIOP
   23.1 How CORBA Works
   23.2 The Bank Example in CORBA
   23.3 A Quick Comparison of CORBA and RMI
   23.4 RMI on Top of CORBA
   23.5 Converting the Bank Example to RMI/IIOP

Colophon

Preface

This book is intended for Java developers who want to build distributed applications. By a
distributed application, I mean a set of programs running in different processes (and quite
possibly on different machines) which form, from the point of view of the end user, a single
application. The latest version of the Java platform, Java 2 (and the associated standard
extension libraries), includes extensive support for building distributed applications.

[1] In this book, program will always refer to Java code executing inside a single Java virtual machine (JVM).
Application, on the other hand, refers to one or more programs executing inside one or more JVMs that, to
the end user, appear to be a single program.
In this book, I will focus on Java's Remote Method Invocation (RMI) framework. RMI is a robust and effective way to build distributed applications in which all the participating programs are written in Java. Because the designers of RMI assumed that all the participating programs would be written in Java, RMI is a surprisingly simple and easy framework to use. Not only is RMI useful for building distributed applications, it is an ideal environment for Java programmers learning how to build a distributed application.

I don't assume you know anything about distributed programs or computer networking. We'll start from the ground up and cover all the concepts, classes, and ideas underlying RMI. I will also cover some of the more advanced aspects of Java programming; it would be irresponsible to write a book on RMI without devoting some space to topics such as sockets and threading.

In order to get the most out of this book, you will need a certain amount of experience with the Java programming language. You should be comfortable programming in Java; you should have a system with which you can experiment with the code examples (like many things, distributed programming is best learned by doing); you should be fairly comfortable with the basics of the JDK 1.1 event model (in particular, many of the code examples are action listeners that have been added to a button); and you should be willing to make mistakes along the way.

About This Book

This book covers an enormous amount of ground, starting with streams and sockets and working its way through the basics of building scalable client-server architectures using RMI.

While the order of chapters is a reasonable one, and one that has served me well in introducing RMI to my students at U.C. Berkeley Extension, it is nonetheless the case that skipping around can sometimes be beneficial. For example, Chapter 10, which discusses object serialization, really relies only on streams (from Chapter 1) and can profitably be read immediately after Chapter 4 (where the first RMI application is introduced).

The book is divided into three sections. Part I starts with an introduction to some of the essential background material for RMI. After presenting the basics of Java's stream and socket libraries, we build a simple socket-based distributed application and then rebuild this application using RMI. At this point, we've actually covered most of the basics of building a simple RMI application.

The rest of Part I (Chapters Chapter 5 through Chapter 9) presents a fairly detailed analysis of how introducing a network changes the various aspects of application design. These chapters culminate in a set of principles for partitioning an application into clients and servers and for designing client-server interaction. Additionally, they introduce an example from banking which is referred to repeatedly in the remainder of the book. After finishing the first section, you will be able to design and build simple RMI applications that, while not particularly scalable or robust, can be used in a variety of situations.

Part II builds on the first by drilling down on the underlying technologies and discussing the implementation decisions that must be made in order to build scalable and secure distributed applications. That is, the first section focuses on the design issues associated with the client-server boundary, and the second section discusses how to make the server scale. As such, this section is less about RMI, or the network interface, and more about how to use the underlying Java technologies (e.g., how to use threads). These chapters can be tough sledding—this is the technical heart of the book.

Part III consists of a set of independent chapters discussing various advanced features of RMI. The distinction between the second and third sections is that everything covered in the second section is essential material for building a sophisticated RMI application (and hence should be at least partially understood by any programmer involved in the design or implementation of an RMI
application). The topics covered in Part III are useful and important for many applications but are not essential knowledge.

What follows is a more detailed description of each chapter in this book.

**Part I**

**Chapter 1**

Streams are a fairly simple data structure; they are best thought of as linear sequences of bytes. They are commonly used to send information to devices (such as a hard drive) or over a network. This chapter is a background chapter that covers Java’s support for streams. It is not RMI-specific at all.

**Chapter 2**

Sockets are a fairly common abstraction for establishing and maintaining a network connection between two programs. Socket libraries exist in most programming languages and across most operating systems. This chapter is a background chapter which covers Java’s socket classes. It is not RMI-specific at all.

**Chapter 3**

This chapter is an exercise in applying the contents of the first two chapters. It uses sockets (and streams) to build a distributed application. Consequently, many of the fundamental concepts and problems of distributed programming are introduced. Because this chapter relies only on the contents of the first two chapters, these concepts and problems are stated with minimal terminology.

**Chapter 4**

This chapter contains a translation of the socket-based printer server into an RMI application. Consequently, it introduces the basic features of RMI and discusses the necessary steps when building a simple RMI application. This is the first chapter in the book that actually uses RMI.

**Chapter 5**

The bank example is one of the oldest and hoariest examples in client-server computing. Along with the printer example, it serves as a running example throughout the book.

**Chapter 6**

The first step in designing and building a typical distributed application is figuring out what the servers are. That is, finding which functionality is in the servers, and deciding how to partition this functionality across servers. This chapter contains a series of guidelines and questions that will help you make these decisions.

**Chapter 7**

Once you've partitioned an application, by placing some functionality in various servers and some functionality in a client, you then need to specify how these components will talk to each other. In other words, you need to design a set of interfaces. This chapter contains a series of guidelines and questions that will help you design and evaluate the interfaces on your servers.

**Chapter 8**

After the heady abstractions and difficult concepts of the previous two chapters, this chapter is a welcome dive into concrete programming tasks. In it, we give the first (of many!) implementations of the bank example, reinforcing the lessons of Chapter 4 and discussing some of the basic implementation decisions that need to be made on the server side.

**Chapter 9**
The final chapter in the first section rounds out the implementation of the bank example. In it, we build a simple client application and the launch code (the code that starts the servers running and makes sure the clients can connect to the servers).

**Part II**

**Chapter 10**

Serialization is the algorithm that RMI uses to encode information sent over the wire. It's easy to use serialization, but using it efficiently and effectively takes a little more work. This chapter explains the serialization mechanism in gory detail.

**Chapter 11**

This is the first of two chapters about threading. It covers the basics of threading: what threads are and how to perform basic thread operations in Java. As such, it is not RMI-specific at all.

**Chapter 12**

In this chapter, we take the terminology and operations from Chapter 11 and apply them to the banking example. We do this by discussing a set of guidelines for making applications multithreaded and then apply each guideline to the banking example. After this, we'll discuss pools, which are a common idiom for reusing scarce resources.

**Chapter 13**

This chapter covers the tenets of testing a distributed application. While these tenets are applied to the example applications from this book, they are not inherently RMI-specific. This chapter is simply about ensuring a reasonable level of performance in a distributed application.

**Chapter 14**

The RMI registry is a simple naming service that ships with the JDK. This chapter explores the RMI registry in detail and uses the discussion as a springboard to a more general discussion of how to use a naming service.

**Chapter 15**

This chapter builds on the previous chapter and offers a general discussion of naming services. At the heart of the chapter is an implementation of a much more scalable, flexible, and federated naming service. The implementation of this new naming service is combined with discussions of general naming-service principles and also serves as another example of how to write code with multiple threads in mind. This chapter is by far the most difficult in the book and can safely be skipped on a first reading.

**Chapter 16**

There's an awful lot of code that handles the interactions between the client and the server. There doesn't seem to be a generally approved name for this code, but I call it the "RMI Runtime." The RMI Runtime handles the details of maintaining connections and implements distributed garbage collection. In this chapter, we'll discuss the RMI Runtime and conclude with an examination of many of the basic system parameters that can be used to configure the RMI Runtime.

**Chapter 17**

The final chapter in Part II deals with a common design pattern called "The Factory Pattern" (or, more typically, "Factories"). After discussing this pattern, we'll dive into the Activation Framework. The Activation Framework greatly simplifies the implementation of The Factory Pattern in RMI.

**Part III**
Chapter 18
RMI is a framework for distributing the objects in an application. It relies, quite heavily, on the socket classes discussed in Chapter 2. However, precisely which type of socket used by an RMI application is configurable. This chapter covers how to switch socket types in an RMI application.

Chapter 19
Dynamic class loading allows you to automatically update an application by downloading .class files as they are needed. It’s one of the most innovative features in RMI and a frequent source of confusion.

Chapter 20
One of the biggest changes in Java 2 was the addition of a full-fledged (and rather baroque) set of security classes and APIs. Security policies are a generalization of the applet “sandbox” and provide a way to grant pieces of code permission to perform certain operations (such as writing to a file).

Chapter 21
Up until this chapter, all the complexity has been on the server side of the application. There’s a good reason for this—the complexity on the client side often involves the details of Swing programming and not RMI. But sometimes, you need to build a more sophisticated client. This chapter discusses when it is appropriate to do so, and covers the basic implementation strategies.

Chapter 22
Firewalls are a reality in today’s corporate environment. And sometimes, you have to tunnel through them. This chapter, which is the most “cookbooky” chapter in the book, tells you how to do so.

Chapter 23
This chapter concerns interoperability with CORBA. CORBA is another framework for building distributed applications; it is very similar to RMI but has two major differences: it is not Java-specific, and the CORBA specification is controlled by an independent standards group (not by Sun Microsystems, Inc.). These two facts make CORBA very popular. After briefly discussing CORBA, this chapter covers RMI/IIOP, which is a way to build RMI applications that “speak CORBA.”

About the Example Code
This book comes with a lot of example code. The examples were written in Java 2, using JDK1.3. While the fundamentals of RMI have not changed drastically from earlier versions of Java, there have been some changes. As a result, you will probably experience some problems if you try and use the example code with earlier versions of Java (e.g., JDK1.1.*).

In addition, you should be aware that the name RMI is often used to refer to two different things. It refers to a set of interfaces and APIs that define a framework for distributed programming. But it also refers to the implementation of those interfaces and APIs written by Javasoft and bundled as part of the JDK. The intended meaning is usually clear from the context. But you should be aware that there are other implementations of the RMI interfaces (most notably from BEA/Weblogic), and that some of the more advanced examples in this book may not work with implementations other than Javasoft’s.

Please don’t use the code examples in this book in production applications. The code provided is example code; it is intended to communicate concepts and explain ideas. In particular, the example code is not particularly robust code. Exceptions are often caught silently and
clauses are rare. Including industrial strength example code would have made the book much longer and the examples more difficult to understand.

**Conventions Used in This Book**

*Italic* is used for:

- Pathnames, filenames, directories, and program names
- New terms where they are defined
- Internet addresses, such as domain names and URLs

*Constant Width* is used for:

- Anything that appears literally in a Java program, including keywords, datatypes, constants, method names, variables, classnames, and interface names
- Command lines and options that should be typed verbatim on the screen
- All JSP and Java code listings
- HTML documents, tags, and attributes

*Constant Width Italic* is used for:

- General placeholders that indicate that an item should be replaced by some actual value in your own program

*Constant width bold* is used for:

- Text that is typed in code examples by the user

---

This icon designates a note, which is an important aside to the nearby text.

This icon designates a warning relating to the nearby text.

---

**Coding Conventions**

For the most part, the examples are written in a fairly generic coding style. I follow standard Java conventions with respect to capitalization. Instance variables are preceded by an underscore (_), while locally scoped variables simply begin with a lowercase letter.

Variable and method names are longer, and more descriptive, than is customary. References to methods within the body of a paragraph almost always omit arguments—instead of `readFromStream(InputStream inputStream)`, we usually write `readFromStream()`.

We will occasionally discuss automatically generated code such as that produced by the RMI compiler. This code is harder to read and often contains variables with names like `$param_DocumentDescription_1`.

Occasionally, an ellipsis will show up in the source code listings. Lines such as:

```java
catch (PrinterException printerException){
```
simply indicate that some uninteresting or irrelevant code has been omitted from the listings in the book.

The class definitions all belong to subpackages of `com.ora.rmibook`. Each chapter of this book has its own package—the examples for **Chapter 1** are contained in subpackages of `com.ora.rmibook.chapter1`; the examples for **Chapter 2** are contained in subpackages of `com.ora.rmibook.chapter2`, and so on. I have tried to make the code for each chapter complete in and of itself. That is, the code for **Chapter 4** does not reference the code for **Chapter 3**. This makes it a little easier to browse the source code and to try out the individual projects. But, as a result of this, there is a large amount of duplication in the example code (many of the classes appear in more than one chapter).

I have also avoided the use of anonymous or local inner classes (while useful, they tend to make code more difficult to read). In short, if you can easily read, and understand, the following snippet:

```java
private void buildGUI() {
    JPanel mainPanel = new JPanel(new BorderLayout);
    _messageBox = new JTextArea;
    mainPanel.add(new JScrollPane(_messageBox), BorderLayout.CENTER);
    createButtons();
}
```

you should have no problem following along with the example code for this book.

**Applications**

The source code for this book is organized as a set of example applications. In order to make it easier to browse the code base, I've tried to follow a consistent naming convention for classes that contain a `main()` method. If a class `Foo` contains a `main()` method, then there will be a companion class `FooFrame` in the same package as `Foo`. Thus, for example, the `ViewFile` application from **Chapter 1** has a companion class `ViewFileFrame`. In fact, `ViewFile` consists entirely of the following code:

```java
package com.ora.rmibook.section1.chapter1;

public class ViewFile {
    public static void main(String[] arguments) {
        (new ViewFileFrame()).show();
    }
}
```

Having top-level GUI classes end in `Frame` makes it a little easier to browse the code in an IDE. For example, **Figure P-1** shows a screenshot of JBuilder 3.0, displaying the source files related to **Chapter 2**.

**Figure P-1. Screenshot of JBuilder 3.0**