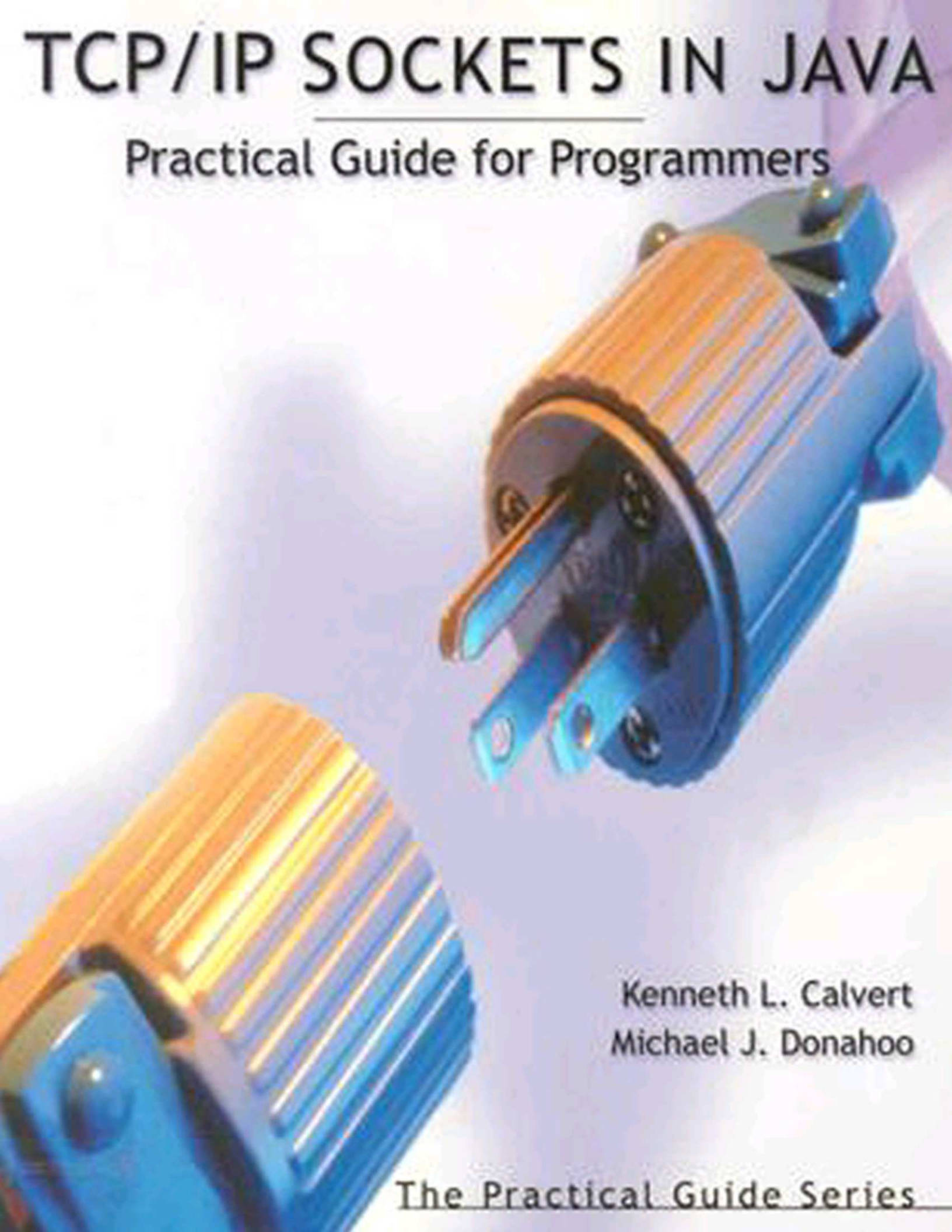
TCP/IP SOCKETS IN JAVA

Practical Guide for Programmers



Kenneth L. Calvert
Michael J. Donahoo

The Practical Guide Series

TCP/IP Sockets in Java:
Practical Guide for Programmers

The Morgan Kaufmann Practical Guides Series
Series Editor: Michael J. Donahoo

TCP/IP Sockets in Java: Practical Guide for Programmers
Kenneth L. Calvert and Michael J. Donahoo

JDBC: Practical Guide for Java Programmers
Gregory D. Speegle

TCP/IP Sockets in C: Practical Guide for Programmers
Michael J. Donahoo and Kenneth L. Calvert

For further information on these books and for a list of forthcoming titles, please visit our Web site at *www.mkp.com*.

TCP/IP Sockets in Java: Practical Guide for Programmers

Kenneth L. Calvert

University of Kentucky

Michael J. Donahoo

Baylor University

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ACADEMIC PRESS

A Division of Harcourt, Inc.

SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

Senior Editor Rick Adams
Publishing Services Manager Scott Norton
Senior Production Editor Cheri Palmer
Assistant Acquisitions Editor Karyn Johnson
Production Coordinator Mei Levenson
Cover Design Matt Seng/Seng Design
Cover Image The Image Bank/Michel Tcherevkoff Ltd.
Text Design Mark Ong/Side by Side Studios
Technical Illustration/Composition Windfall Software, using ZzTEX
Copyeditor Sharilyn Hovind
Proofreader Jennifer McClain
Indexer Bill Meyers
Printer Edwards Brothers

Morgan Kaufmann Publishers
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
http://www.mkp.com

ACADEMIC PRESS
A Division of Harcourt, Inc.
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA
http://www.academicpress.com

Academic Press
Harcourt Place, 32 Jamestown Road, London, NW1 7BY, United Kingdom
http://www.academicpress.com

Printed in the United States of America

06 05 04 03 02 5 4 3 2 1

Library of Congress Control Number: 2001094369
ISBN: 1-55860-685-8

This book is printed on acid-free paper.

To Tricia and Lisa.

This Page Intentionally Left Blank

Contents

Preface

For years, college courses in computer networking were taught with little or no hands on experience. For various reasons, including some good ones, instructors approached the principles of computer networking primarily through equations, analyses, and abstract descriptions of protocol stacks. Textbooks might have included code, but it would have been unconnected to anything students could get their hands on. We believe, however, that students learn better when they can see (and then build) concrete examples of the principles at work. And, fortunately, things have changed. The Internet has become a part of everyday life, and access to its services is readily available to most students (and their programs). Moreover, copious examples—good and bad—of nontrivial software are freely available.

We wrote this book for the same reason we wrote *TCP/IP Sockets in C*: we needed a resource to support learning networking through programming exercises in our courses. Our goal is to provide a sufficient introduction so that students can get their hands on real network services without too much hand-holding. After grasping the basics, students can then move on to more advanced assignments, which support learning about routing algorithms, multimedia protocols, medium access control, and so on. We have tried to make this book equivalent to our earlier book to enable instructors to allow students to choose the language they use and still ensure that all students will come away with the same skills and understanding. Of course, it is not clear that this goal is achievable, but in any case the scope, price, and presentation level of the book are intended to be similar.

Intended Audience

This book is aimed primarily at students in upper-division undergraduate or graduate courses in computer networks. It is intended as a supplement to a traditional textbook that explains the problems and principles of computer networks. At the same time, we have tried to make the

book reasonably self-contained (except for the assumed programming background), so that it can also be used, for example, in courses on operating systems or distributed computing. For uses outside the context of a networking course, it will be helpful if the students have some acquaintance with the basic concepts of networking and TCP/IP.

This book's other target audience consists of practitioners who know Java and want to learn about writing Java applications that use TCP/IP. This book should take such users far enough that they can start experimenting and learning on their own. Readers are assumed to have access to a computer equipped with Java. This book is based on Version 1.3 of Java and the Java Virtual Machine (JVM); however, the code should work with earlier versions of Java, with the exception of a few new Java methods. Java is about portability, so the particular hardware and operating system (OS) on which you run should not matter.

Approach

Chapter 1 provides a general overview of networking concepts. It is not, by any means, a complete introduction, but rather is intended to allow readers to synchronize with the concepts and terminology used throughout the book. Chapter 2 introduces the mechanics of simple clients and servers; the code in this chapter can serve as a starting point for a variety of exercises. Chapter 3 covers the basics of message construction and parsing. The reader who digests the first three chapters should in principle be able to implement a client and server for a given (simple) application protocol. Chapter 4 then deals with techniques that are necessary when building more sophisticated and robust clients and servers. Finally, in keeping with our goal of illustrating principles through programming, Chapter 5 discusses the relationship between the programming constructs and the underlying protocol implementations in somewhat more detail.

Our general approach introduces programming concepts through simple program examples accompanied by line-by-line commentary that describes the purpose of every part of the program. This lets you see the important objects and methods as they are used in context. As you look at the code, you should be able to understand the purpose of each and every line.

Java makes many things easier, but it does not support some functionality that is commonly associated with the C/UNIX sockets interface (asynchronous I/O, select( )-style multiplexing). In C and C++, the socket interface is a generic application programming interface (API) for all types of protocols, not just TCP/IP. Java's socket classes, on the other hand, by default work exclusively with TCP and UDP over IPv4. Ironically, there does not seem to be anything in the Java specification or documentation that requires that an instance of the `Socket` class use TCP, or that a `DatagramSocket` instance use UDP. Nevertheless, this book assumes this to be the case, as is true of current implementations.

Our examples do not take advantage of all library facilities in Java. Some of these facilities, in particular serialization, effectively require that all communicating peers be implemented in Java. Also, to introduce examples as soon as possible, we wanted to avoid bringing in a thicket of methods and classes that have to be sorted out later. We have tried to keep it simple, especially in the early chapters.

What This Book Is Not

To keep the price of this book within a reasonable range for a supplementary text, we have had to limit its scope and maintain a tight focus on the goals outlined above. We omitted many topics and directions, so it is probably worth mentioning some of the things this book is not:

- It is not an introduction to Java. We focus specifically on TCP/IP socket programming using the Java language. We expect that the reader is already acquainted with the language and basic Java libraries (especially I/O), and knows how to develop programs in Java.
- It is not a book on protocols. Reading this book will not make you an expert on IP, TCP, FTP, HTTP, or any other existing protocol (except maybe the echo protocol). Our focus is on the interface to the TCP/IP services provided by the socket abstraction. (It will help if you start with some idea about the general workings of TCP and IP, but Chapter 1 may be an adequate substitute.)
- It is not a guide to all of Java's rich collection of libraries that are designed to hide communication details (e.g., `HTTPConnection`) and make the programmer's life easier. Since we are teaching the fundamentals of how to do, not how to avoid doing, protocol development, we do not cover these parts of the API. We want readers to understand protocols in terms of what goes on the wire, so we mostly use simple byte streams and deal with character encodings explicitly. As a consequence, this text does not deal with `URL`, `URLConnection`, and so on. We believe that once you understand the principles, using these convenience classes will be straightforward. The network-relevant classes that we do cover include `InetAddress`, `Socket`, `ServerSocket`, `DatagramPacket`, `DatagramSocket`, and `MulticastSocket`.
- It is not a book on object-oriented design. Our focus is on the important principles of TCP/IP socket programming, and our examples are intended to illustrate them concisely. As far as possible, we try to adhere to object-oriented design principles; however, when doing so adds complexity that obfuscates the socket principles or bloats the code, we sacrifice design for clarity. This text does not cover design patterns for networking. (Though we would like to think that it provides some of the background necessary for understanding such patterns!)
- It is not a book on writing production-quality code. Again, though we strive for robustness, the primary goal of our code examples is education. In order to avoid obscuring the principles with large amounts of error-handling code, we have sacrificed some robustness for brevity and clarity.
- It is not a book on doing your own native sockets implementation in Java. We focus exclusively on TCP/IP sockets as provided by the standard Java distribution and do not cover the various socket implementation wrapper classes (e.g., `SocketImpl`).
- To avoid cluttering the examples with extraneous (nonsocket-related programming) code, we have made them command-line based. While the book's Web site, *www.mkp.com/practical/javasockets,* contains a few examples of GUI-enhanced network applications, we do not include or explain them in this text.

- It is not a book on Java applets. Applets use the same Java networking API so the communication code should be very similar; however, there are severe security restrictions on the kinds of communication an applet can perform. We provide a very limited discussion of these restrictions and a single applet/application example on the Web site; however, a complete description of applet networking is beyond the scope of this text.

This book will not make you an expert—that takes years of experience. However, we hope it will be useful as a resource, even to those who already know quite a bit about using sockets in Java. Both of us enjoyed writing it and learned quite a bit along the way.

Acknowledgments

We would like to thank all the people who helped make this book a reality. Despite the book's brevity, many hours went into reviewing the original proposal and the draft, and the reviewers' input has significantly shaped the final result.

First, thanks to those who meticulously reviewed the draft of the text and made suggestions for improvement. These include Michel Barbeau, Carlton University; Chris Edmondson-Yurkanan, University of Texas at Austin, Ted Herman, University of Iowa; Dave Hollinger, Rensselaer Polytecnic Institute; Jim Leone, Rochester Institute of Technology; Dan Schmidt, Texas A&M University; Erick Wagner, EDS; and CSI4321, Spring 2001. Any errors that remain are, of course, our responsibility. We are very interested in weeding out such errors in future printings so if you find one, please email either of us. We will maintain an errata list on the book's Web page.

Finally, we are grateful to the folks at Morgan Kaufmann. They care about quality and we appreciate that. We especially appreciate the efforts of Karyn Johnson, our editor, and Mei Levenson, our production coordinator.

Feedback

We invite your suggestions for the improvement of any aspect of this book. You can send feedback via the book's Web page, *www.mkp.com/practical/javasockets,* or you can email us at the addresses below:

Kenneth L. Calvert *calvert@netlab.uky.edu*

Michael J. Donahoo *Jeff_Donahoo@baylor.edu*

chapter 1

Introduction

Millions of computers all over the world are now connected to the worldwide network known as the Internet. The Internet enables programs running on computers thousands of miles apart to communicate and exchange information. If you have a computer connected to a network, you may have used a Web browser—a typical program that makes use of the Internet. What does such a program do to communicate with others over a network? The answer varies with the application and the operating system (OS), but a great many programs get access to network communication services through the sockets application programming interface (API). The goal of this book is to get you started writing Java programs that use the sockets API.

Before delving into the details of the API, it is worth taking a brief look at the big picture of networks and protocols to see how an API for Transmission Control Protocol/Internet Protocol fits in. Our goal here is not to teach you how networks and TCP/IP work—many fine texts are available for that purpose [2, 4, 11, 16, 22]—but rather to introduce some basic concepts and terminology.

1.1 Networks, Packets, and Protocols

A computer network consists of machines interconnected by communication channels. We call these machines *hosts* and *routers*. Hosts are computers that run applications such as your Web browser. The application programs running on hosts are really the users of the network. Routers are machines whose job is to relay, or *forward*, information from one communication channel to another. They may run programs but typically do not run application programs. For our purposes, a *communication channel* is a means of conveying sequences of bytes from one host to another; it may be a broadcast technology like Ethernet, a dial-up modem connection, or something more sophisticated.

Routers are important simply because it is not practical to connect every host directly to every other host. Instead, a few hosts connect to a router, which connects to other routers, and so on to form the network. This arrangement lets each machine get by with a relatively

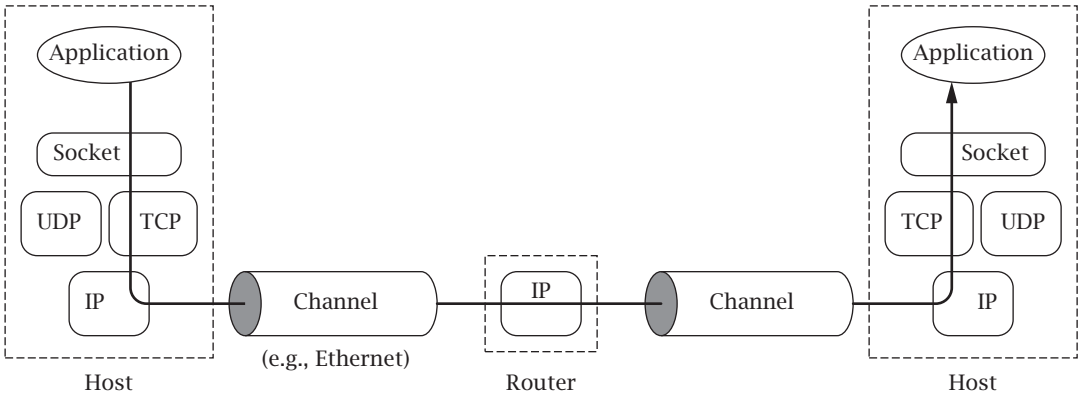


Figure 1.1: A TCP/IP network.

small number of communication channels; most hosts need only one. Programs that exchange information over the network, however, do not interact directly with routers and generally remain blissfully unaware of their existence.

By *information* we mean sequences of bytes that are constructed and interpreted by programs. In the context of computer networks, these byte sequences are generally called *packets*. A packet contains control information that the network uses to do its job and sometimes also includes user data. An example is information identifying the packet's destination. Routers use such control information to figure out how to forward each packet.

A *protocol* is an agreement about the packets exchanged by communicating programs and what they mean. A protocol tells how packets are structured—for example, where the destination information is located in the packet and how big it is—as well as how the information is to be interpreted. A protocol is usually designed to solve a specific problem using given capabilities. For example, the *HyperText Transfer Protocol (HTTP)* solves the problem of transferring hypertext objects between servers, where they are stored, and Web browsers that make them available to human users.

Implementing a useful network requires that a large number of different problems be solved. To keep things manageable and modular, different protocols are designed to solve different sets of problems. TCP/IP is one such collection of solutions, sometimes called a *protocol suite*. It happens to be the suite of protocols used in the Internet, but it can be used in stand-alone private networks as well. Henceforth when we talk about the "network," we mean any network that uses the TCP/IP protocol suite. The main protocols in the TCP/IP suite are the Internet Protocol (IP), the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP).

It turns out to be useful to organize protocols into *layers*; TCP/IP and virtually all other protocol suites are organized this way. Figure 1.1 shows the relationships among the protocols, applications, and the sockets API in the hosts and routers, as well as the flow of data from one application (using TCP) to another. The boxes labeled TCP, UDP, and IP represent implementations of those protocols. Such implementations typically reside in the

operating system of a host. Applications access the services provided by UDP and TCP through the sockets API. The arrow depicts the flow of data from the application, through the TCP and IP implementations, through the network, and back up through the IP and TCP implementations at the other end.

In TCP/IP, the bottom layer consists of the underlying communication channels—for example, Ethernet or dial-up modem connections. Those channels are used by the *network layer*, which deals with the problem of forwarding packets toward their destination (i.e., what routers do). The single network layer protocol in the TCP/IP suite is the Internet Protocol; it solves the problem of making the sequence of channels and routers between any two hosts look like a single host-to-host channel.

The Internet Protocol provides a *datagram* service: every packet is handled and delivered by the network independently, like letters or parcels sent via the postal system. To make this work, each IP packet has to contain the *address* of its destination, just as every package that you mail is addressed to somebody. (We'll say more about addresses shortly.) Although most delivery companies guarantee delivery of a package, IP is only a best-effort protocol: it attempts to deliver each packet, but it can (and occasionally does) lose, reorder, or duplicate packets in transit through the network.

The layer above IP is called the *transport layer*. It offers a choice between two protocols: TCP and UDP. Each builds on the service provided by IP, but they do so in different ways to provide different kinds of transport, which are used by *application protocols* with different needs. TCP and UDP have one function in common: addressing. Recall that IP delivers packets to hosts; clearly, a finer granularity of addressing is needed to get a packet to a particular application, perhaps one of many using the network on the same host. Both TCP and UDP use addresses, called *port numbers*, to identify applications within hosts. They are called *end-to-end transport protocols* because they carry data all the way from one program to another (whereas IP only carries data from one host to another).

TCP is designed to detect and recover from the losses, duplications, and other errors that may occur in the host-to-host channel provided by IP. TCP provides a *reliable byte-stream* channel, so that applications do not have to deal with these problems. It is a *connection-oriented* protocol: before using it to communicate, two programs must first establish a TCP connection, which involves completing an exchange of *handshake messages* between the TCP implementations on the two communicating computers. Using TCP is also similar in many ways to file input/output (I/O). In fact, a file that is written by one program and read by another is a reasonable model of communication over a TCP connection. UDP, on the other hand, does not attempt to recover from errors experienced by IP; it simply extends the IP best-effort datagram service so that it works between application programs instead of between hosts. Thus, applications that use UDP must be prepared to deal with losses, reordering, and so on.

1.2 About Addresses

When you mail a letter, you provide the address of the recipient in a form that the postal service can understand. Before you can talk to someone on the phone, you must supply their number to the telephone system. In a similar way, before a program can communicate with

another program, it must tell the network where to find the other program. In TCP/IP, it takes two pieces of information to identify a particular program: an *Internet address*, used by IP, and a *port number*, the additional address interpreted by the transport protocol (TCP or UDP).

Internet addresses are 32-bit binary numbers.[1] In writing down Internet addresses for human consumption (as opposed to using them inside applications), we typically show them as a string of four decimal numbers separated by periods (e.g., 10.1.2.3); this is called the *dotted-quad* notation. The four numbers in a dotted-quad string represent the contents of the four bytes of the Internet address—thus, each is a number between 0 and 255.

One special IP address worth knowing is the *loopback address*, 127.0.0.1. This address is always assigned to a special *loopback interface*, which simply echoes transmitted packets right back to the sender. The loopback interface is very useful for testing; it can be used even when a computer is not connected to the network.

Technically, each Internet address refers to the connection between a host and an underlying communication channel, such as a dial-up modem or Ethernet card. Because each such network connection belongs to a single host, an Internet address identifies a host as well as its connection to the network. However, because a host can have multiple physical connections to the network, one host can have multiple Internet addresses.

The port number in TCP or UDP is always interpreted relative to an Internet address. Returning to our earlier analogies, a port number corresponds to a room number at a given street address, say, that of a large building. The postal service uses the street address to get the letter to a mailbox; whoever empties the mailbox is then responsible for getting the letter to the proper room within the building. Or consider a company with an internal telephone system: to speak to an individual in the company, you first dial the company's main phone number to connect to the internal telephone system and then dial the extension of the particular telephone of the individual that you wish to speak with. In these analogies, the Internet address is the street address or the company's main number, whereas the port corresponds to the room number or telephone extension. Port numbers are 16-bit unsigned binary numbers, so each one is in the range 1 to 65,535 (0 is reserved).

1.3 About Names

Most likely you are accustomed to referring to hosts by *name* (e.g., host.example.com). However, the Internet protocols deal with numerical addresses, not names. You should understand that the use of names instead of addresses is a convenience feature that is independent of the basic service provided by TCP/IP—you can write and use TCP/IP applications without ever

[1] Throughout this book the term *Internet address* refers to the addresses used with the current version of IP, which is version 4 [12]. Because it is expected that a 32-bit address space will be inadequate for future needs, a new version of IP has been defined [5]; it provides the same service but has much bigger Internet addresses (128 bits). IPv6, as the new version is known, has not been widely deployed; the sockets API will require some changes to deal with its much larger addresses [6].

using a name. When you use a name to identify a communication endpoint, the system has to do some extra work to *resolve* the name into an address.

This extra step is often worth it, for a couple of reasons. First, names are generally easier for humans to remember than dotted-quads. Second, names provide a level of indirection, which insulates users from IP address changes. During the writing of this book, the Web server for the publisher of this text, Morgan Kaufmann, changed Internet addresses from 208.164.121.48 to 216.200.143.124. However, because we refer to that Web server as *www.mkp.com* (clearly much easier to remember than 208.164.121.48) and because the change is reflected in the system that maps names to addresses (*www.mkp.com* now resolves to the new Internet address instead of 208.164.121.48), the change is transparent to programs that use the name to access the Web server.

The name-resolution service can access information from a wide variety of sources. Two of the primary sources are the *Domain Name System (DNS)* and local configuration databases. The DNS [9] is a distributed database that maps *domain names* such as *www.mkp.com* to Internet addresses and other information; the DNS protocol [10] allows hosts connected to the Internet to retrieve information from that database using TCP or UDP. Local configuration databases are generally OS-specific mechanisms for local name-to-Internet address mappings.

1.4 Clients and Servers

In our postal and telephone analogies, each communication is initiated by one party, who sends a letter or makes the telephone call, while the other party responds to the initiator's contact by sending a return letter or picking up the phone and talking. Internet communication is similar. The terms *client* and *server* refer to these roles: The client program initiates communication, while the server program waits passively for and then responds to clients that contact it. Together, the client and server compose the *application*. The terms *client* and *server* are descriptive of the typical situation in which the server makes a particular capability—for example, a database service—available to any client that is able to communicate with it.

Whether a program is acting as a client or server determines the general form of its use of the sockets API to establish communication with its *peer*. (The client is the peer of the server and vice versa.) Beyond that, the client-server distinction is important because the client needs to know the server's address and port initially, but not vice versa. With the sockets API, the server can, if necessary, learn the client's address information when it receives the initial communication from the client. This is analogous to a telephone call—in order to be called, a person does not need to know the telephone number of the caller. As with a telephone call, once the connection is established, the distinction between server and client disappears.

How does a client find out a server's IP address and port number? Usually, the client knows the name of the server it wants—for example, from a *Universal Resource Locator (URL)* such as *http://www.mkp.com*—and uses the name-resolution service to learn the corresponding Internet address.

Finding a server's port number is a different story. In principle, servers can use any port, but the client must be able to learn what it is. In the Internet, there is a convention of assigning well-known port numbers to certain applications. The Internet Assigned Number Authority

(IANA) oversees this assignment. For example, port number 21 has been assigned to the *File Transfer Protocol (FTP)*. When you run an FTP client application, it tries to contact the FTP server on that port by default. A list of all the assigned port numbers is maintained by the numbering authority of the Internet (see *http://www.iana.org/assignments/port-numbers*).

1.5 What Is a Socket?

A *socket* is an abstraction through which an application may send and receive data, in much the same way as an open file handle allows an application to read and write data to stable storage. A socket allows an application to plug in to the network and communicate with other applications that are plugged in to the same network. Information written to the socket by an application on one machine can be read by an application on a different machine and vice versa.

Different types of sockets correspond to different underlying protocol suites and different stacks of protocols within a suite. This book deals only with the TCP/IP protocol suite. The main types of sockets in TCP/IP today are *stream sockets* and *datagram sockets*. Stream sockets use TCP as the end-to-end protocol (with IP underneath) and thus provide a reliable byte-stream service. A TCP/IP stream socket represents one end of a TCP connection. Datagram sockets use UDP (again, with IP underneath) and thus provide a best-effort datagram service that applications can use to send individual messages up to about 65,500 bytes in length. Stream and datagram sockets are also supported by other protocol suites, but this book deals only with TCP stream sockets and UDP datagram sockets. A TCP/IP socket is uniquely identified by an Internet address, an end-to-end protocol (TCP or UDP), and a port number. As you proceed, you will encounter several ways for a socket to become bound to an address.

Figure 1.2 depicts the logical relationships among applications, socket abstractions, protocols, and port numbers within a single host. Note that a single socket abstraction can be referenced by multiple application programs. Each program that has a reference to a particular socket can communicate through that socket. Earlier we said that a port identifies an application on a host. Actually, a port identifies a socket on a host. From Figure 1.2, we see that multiple programs on a host can access the same socket. In practice, separate programs that access the same socket would usually belong to the same application (e.g., multiple copies of a Web server program), although in principle they could belong to different applications.

1.6 Exercises

1. Can you think of a real-life example of communication that does not fit the client-server model?
2. To how many different kinds of networks is your home connected? How many support two-way transport?

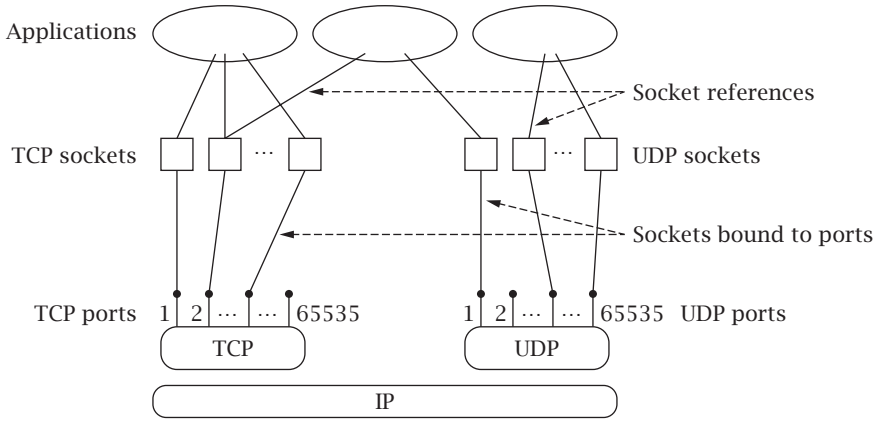


Figure 1.2: Sockets, protocols, and ports.

3. IP is a best-effort protocol, requiring that information be broken down into datagrams, which may be lost, duplicated, or reordered. TCP hides all of this, providing a reliable service that takes and delivers an unbroken stream of bytes. How might you go about providing TCP service on top of IP? Why would anybody use UDP when TCP is available?

This Page Intentionally Left Blank

chapter 2

Basic Sockets

You are now ready to learn about writing your own socket applications. We begin by demonstrating how Java applications identify network hosts. Then, we describe the creation of TCP and UDP clients and servers. Java provides a clear distinction between using TCP and UDP, defining a separate set of classes for both protocols, so we treat each separately.

2.1 Socket Addresses

IP uses 32-bit binary addresses to identify communicating hosts. A client must specify the IP address of the host running the server program when it initiates communication; the network infrastructure uses the 32-bit *destination address* to route the client's information to the proper machine. Addresses can be specified in Java using a string that contains either the dotted-quad representation of the numeric address (e.g., 169.1.1.1) or a name (e.g., *server.example.com*). Java encapsulates the IP addresses abstraction in the `InetAddress` class which provides three static methods for creating `InetAddress` instances. `getByName()` and `getAllByName()` take a name or IP address and return the corresponding `InetAddress` instance(s). For example, `InetAddress.getByName("192.168.75.13")` returns an instance identifying the IP address 192.168.75.13. The third method, `getLocalHost()`, returns an `InetAddress` instance containing the local host address. Our first program example, `InetAddressExample.java`, demonstrates the use of `InetAddress`. The program takes a list of names or IP addresses as command-line parameters and prints the name and an IP address of the local host, followed by names and IP addresses of the hosts specified on the command line.

InetAddressExample.java

```
0  import java.net.*;  // for InetAddress
1
2  public class InetAddressExample {
3
```

```
 4    public static void main(String[] args) {
 5
 6      // Get name and IP address of the local host
 7      try {
 8        InetAddress address = InetAddress.getLocalHost();
 9        System.out.println("Local Host:");
10        System.out.println("\t" + address.getHostName());
11        System.out.println("\t" + address.getHostAddress());
12      } catch (UnknownHostException e) {
13        System.out.println("Unable to determine this host's address");
14      }
15
16      for (int i = 0; i < args.length; i++) {
17        // Get name(s)/address(es) of hosts given on command line
18        try {
19          InetAddress[] addressList = InetAddress.getAllByName(args[i]);
20          System.out.println(args[i] + ":");
21          // Print the first name.  Assume array contains at least one entry.
22          System.out.println("\t" + addressList[0].getHostName());
23          for (int j = 0; j < addressList.length; j++)
24            System.out.println("\t" + addressList[j].getHostAddress());
25        } catch (UnknownHostException e) {
26          System.out.println("Unable to find address for " + args[i]);
27        }
28      }
29    }
30  }
```

InetAddressExample.java

1. **Print information about the local host:** lines 6–14
   - **Create an `InetAddress` instance for the local host:** line 8
   - **Print the local host information:** lines 9–11
     `getHostName()` and `getHostAddress()` return a string for the host name and IP address, respectively.
2. **Request information for each host specified on the command line:** lines 16–28
   - **Create an array of `InetAddress` instances for the specified host:** line 19
     `InetAddress.getAllByName()` returns an array of `InetAddress` instances, one for each of the specified host's addresses.
   - **Print the host information:** lines 22–24

To use this application to find information about the local host and the publisher's Web server (*www.mkp.com*), do the following:

```
% java InetAddressExample www.mkp.com

Local Host:
        tractor.farm.com
        169.1.1.2
www.mkp.com:
        www.mkp.com
        216.200.143.124
```

If we know the IP address of a host (e.g., 169.1.1.1), we find the name of the host by

```
% java InetAddressExample 169.1.1.1

Local Host:
        tractor.farm.com
        169.1.1.2
169.1.1.1:
        base.farm.com
        169.1.1.1
```

When the name service is not available for some reason—say, the program is running on a machine that is not connected to any network—attempting to identify a host by name may fail. Moreover, it may take a significant amount of time to do so, as the system tries various ways to resolve the name to an IP address. It is therefore good to know that you can always refer to a host using the IP address in dotted-quad notation. In any of our examples, if a remote host is specified by name, the host running the example must be configured to convert names to addresses, or the example won't work. If you can ping a host using one of its names (e.g., run the command "ping *server.example.com*"), then the examples should work with names. If your ping test fails or the example hangs, try specifying the host by IP address, which avoids the name-to-address conversion altogether.

InetAddress [1]

Creators

static InetAddress[ ] `getAllByName`(**String** *host*)

Returns the list of addresses for the specified host.

host Host name or address

[1] For each Java networking class described in this text, we present only the primary methods and omit methods that are deprecated or whose use is beyond the scope of this text. As with everything in Java, the specification is a moving target. This information is included to provide an overall picture of the Java socket interface, not as a final authority. We encourage the reader to refer to the API specifications from *java.sun.com* as the current and definitive source.

static InetAddress `getByName`(**String** *host*)
static InetAddress `getLocalHost`()

Returns an IP address for the specified/local host.

host Host name or IP address

Accessors

byte[ ] `getAddress`()

Returns the 4 bytes of the 32-bit IP address in big-endian order.

String `getHostAddress`()

Returns the IP address in dotted-quad notation (e.g., "169.1.1.2").

String `getHostName`()

Returns the canonical name of the host associated with the address.

boolean `isMulticastAddress`()

Returns `true` if the address is a multicast address (see Section 4.3.2).

Operators

boolean `equals`(**Object** *address*)

Returns `true` if *address* is non-null and represents the same address as this `InetAddress` instance.

address Address to compare

2.2 TCP Sockets

Java provides two classes for TCP: `Socket` and `ServerSocket`. An instance of `Socket` represents one end of a TCP connection. A *TCP connection* is an abstract two-way channel whose ends are each identified by an IP address and port number. Before being used for communication, a TCP connection must go through a setup phase, which starts with the client's TCP sending a connection request to the server's TCP. An instance of `ServerSocket` listens for TCP connection requests and creates a new `Socket` instance to handle each incoming connection.

2.2.1 TCP Client

The client initiates communication with a server that is passively waiting to be contacted. The typical TCP client goes through three steps:

1. Construct an instance of `Socket`: The constructor establishes a TCP connection to the specified remote host and port.

2. Communicate using the socket's I/O streams: A connected instance of `Socket` contains an `InputStream` and `OutputStream` that can be used just like any other Java I/O stream (see Chapter 3).
3. Close the connection using the `close()` method of `Socket`.

Our first TCP application, called `TCPEchoClient.java`, is a client that communicates with an *echo server* using TCP. An echo server simply repeats whatever it receives back to the client. The string to be echoed is provided as a command-line argument to our client. Many systems include an echo server for debugging and testing purposes. To test if the standard echo server is running, try telnetting to port 7 (the default echo port) on the server (e.g., at command line "`telnet server.example.com 7`" or use your basic telnet application).

TCPEchoClient.java

```
 0  import java.net.*;  // for Socket
 1  import java.io.*;   // for IOException and Input/OutputStream
 2
 3  public class TCPEchoClient {
 4
 5    public static void main(String[] args) throws IOException {
 6
 7      if ((args.length < 2) || (args.length > 3))  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");
 9
10      String server = args[0];       // Server name or IP address
11      // Convert input String to bytes using the default character encoding
12      byte[] byteBuffer = args[1].getBytes();
13
14      int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
15
16      // Create socket that is connected to server on specified port
17      Socket socket = new Socket(server, servPort);
18      System.out.println("Connected to server...sending echo string");
19
20      InputStream in = socket.getInputStream();
21      OutputStream out = socket.getOutputStream();
22
23      out.write(byteBuffer);  // Send the encoded string to the server
24
25      // Receive the same string back from the server
26      int totalBytesRcvd = 0;  // Total bytes received so far
27      int bytesRcvd;           // Bytes received in last read
28      while (totalBytesRcvd < byteBuffer.length) {
29        if ((bytesRcvd = in.read(byteBuffer, totalBytesRcvd,
30                          byteBuffer.length - totalBytesRcvd)) == -1)
31          throw new SocketException("Connection closed prematurely");
```

```
32        totalBytesRcvd += bytesRcvd;
33      }
34
35      System.out.println("Received: " + new String(byteBuffer));
36
37      socket.close();  // Close the socket and its streams
38    }
39  }
```

TCPEchoClient.java

1. **Application setup and parameter parsing:** lines 0–14
   - **Convert the echo string:** line 12
     TCP sockets send and receive sequences of bytes. The `getBytes()` method of `String` returns a byte array representation of the string. (See Section 3.1 for a discussion of character encodings.)
   - **Determine the port of the echo server:** line 14
     The default echo port is 7. If we specify a third parameter, `Integer.parseInt()` takes the string and returns the equivalent integer value.
2. **TCP socket creation:** line 17
   The `Socket` constructor creates a socket and establishes a connection to the specified server, identified either by name or IP address. Note that the underlying TCP deals only with IP addresses. If a name is given, the implementation resolves it to the corresponding address. If the connection attempt fails for any reason, the constructor throws an `IOException`.
3. **Get socket input and output streams:** lines 20–21
   Associated with each connected `Socket` instance is an `InputStream` and `OutputStream`. We send data over the socket by writing bytes to the `OutputStream` just as we would any other stream, and we receive by reading from the `InputStream`.
4. **Send the string to echo server:** line 23
   The `write()` method of `OutputStream` transmits the given byte array over the connection to the server.
5. **Receive the reply from the echo server:** lines 25–33
   Since we know the number of bytes to expect from the echo server, we can repeatedly receive bytes until we have received the same number of bytes we sent. This particular form of `read()` takes three parameters: 1) buffer to receive into, 2) byte offset into the buffer where the first byte received should be placed, and 3) the maximum number of bytes to be placed in the buffer. `read()` blocks until some data is available, reads up to the specified maximum number of bytes, and returns the number of bytes actually placed in the buffer (which may be less than the given maximum). The loop simply fills

up *byteBuffer* until we receive as many bytes as we sent. If the TCP connection is closed by the other end, `read()` returns −1. For the client, this indicates that the server prematurely closed the socket.

Why not just a single read? TCP does not preserve `read()` and `write()` message boundaries. That is, even though we sent the echo string with a single `write()`, the echo server may receive it in multiple chunks. Even if the echo string is handled in one chunk by the echo server, the reply may still be broken into pieces by TCP. One of the most common errors for beginners is the assumption that data sent by a single `write()` will always be received in a single `read()`.

6. **Print echoed string:** line 35
   To print the server's response, we must convert the byte array to a string using the default character encoding.
7. **Close socket:** line 37
   When the client has finished receiving all of the echoed data, it closes the socket.

We can communicate with an echo server named *server.example.com* with IP address 169.1.1.1 in either of the following ways:

```
% java TCPEchoClient server.example.com "Echo this!"
Received: Echo this!
% java TCPEchoClient 169.1.1.1 "Echo this!"
Received: Echo this!
```

See `TCPEchoClientGUI.java` on the book's Web site for an implementation of the TCP echo client with a graphical interface.

Socket

Constructors

`Socket`(**InetAddress** *remoteAddr*, **int** *remotePort*)

`Socket`(**String** *remoteHost*, **int** *remotePort*)

`Socket`(**InetAddress** *remoteAddr*, **int** *remotePort*, **InetAddress** *localAddr*, **int** *localPort*)

`Socket`(**String** *remoteHost*, **int** *remotePort*, **InetAddress** *localAddr*, **int** *localPort*)

Constructs a TCP socket connected to the specified remote address and port. The first two forms of the constructor do not specify the local address and port, so a default local address and some available port are chosen. Specifying the local address may be useful on a host with multiple interfaces.

remoteAddr	Remote host address
remoteHost	Remote host name or IP address (in dotted-quad form)
remotePort	Remote port

localAddr	Local address; use `null` to specify using the default local address
localPort	Local port; a localPort of 0 allows the constructor to pick any available port

Operators

void `close()`

Closes the TCP socket and its I/O streams.

void `shutdownInput()`

Closes the input side of a TCP stream. Any unread data is silently discarded, including data buffered by the socket, data in transit, and data arriving in the future. Any subsequent attempt to read from the socket will return end-of-stream (−1); any subsequent call to `getInputStream()` will cause an `IOException` to be thrown (see Section 4.5).

void `shutdownOutput()`

Closes the output side of a TCP stream. The implementation will attempt to deliver any data already written to the socket's output stream to the other end. Any subsequent attempt to write to the socket's output stream or to call `getOutputStream()` will cause an `IOException` to be thrown (see Section 4.5).

Accessors/Mutators

InetAddress `getInetAddress()`

int `getPort()`

Returns the remote socket address/port.

InputStream `getInputStream()`

OutputStream `getOutputStream()`

Returns a stream for reading/writing bytes from/to the socket.

boolean `getKeepAlive()`

void `setKeepAlive(`**boolean** *on*`)`

Returns/sets keepalive message behavior. If keepalive is enabled, TCP sends a probe to the other end of the connection when no data has been exchanged for a system-dependent amount of time (usually two hours). If the remote socket is still alive, it will acknowledge the probe (invisible to the application). However, if the other end fails to acknowledge several probes in a row, the local TCP closes the connection, and subsequent operations on it will throw an exception. Keepalive is disabled by default.

on	If `true` (`false`), enable (disable) keepalive.

InetAddress `getLocalAddress()`
int `getLocalPort()`

Returns the local socket address/port.

int `getReceiveBufferSize()`
int `getSendBufferSize()`
void `setReceiveBufferSize(`**int** *size*`)`
void `setSendBufferSize(`**int** *size*`)`

Returns/sets the size of the send/receive buffer for the socket (see Section 4.4).

size Number of bytes to allocate for the socket send/receive buffer

int `getSoLinger()`
void `setSoLinger(`**boolean** *on*, **int** *linger*`)`

Returns/sets the maximum amount of time (in milliseconds) that `close()` will block waiting for all data to be delivered. `getSoLinger()` returns −1 if lingering is disabled (see Section 5.4). Lingering is off by default.

on If `true`, the socket lingers on `close()`. up to the maximum specified time.

linger The maximum amount of time (milliseconds) a socket lingers on `close()`

int `getSoTimeout()`
void `setSoTimeout(`**int** *timeout*`)`

Returns/sets the maximum amount of time that a `read()` on this socket will block. If the specified number of milliseconds elapses before any data is available, an `InterruptedIOException` is thrown (see Section 4.2).

timeout The maximum time (milliseconds) to wait for data on a `read()`. The value 0 (the default) indicates that there is no time limit, meaning that a read will not return until data is available.

boolean `getTcpNoDelay()`
void `setTcpNoDelay(`**boolean** *on*`)`

Returns/sets whether the Nagle algorithm to coalesce TCP packets is disabled. To avoid small TCP packets, which make inefficient use of network resources, Nagle's algorithm (enabled by default) delays packet transmission under certain conditions to improve the opportunities to coalesce bytes from several writes into a single TCP packet. This delay is unacceptable to some types of interactive applications.

on If `true` (`false`), disable (enable) Nagle's algorithm.

Caveat: By default, `Socket` is implemented on top of a TCP connection; however, in Java, you can actually change the underlying implementation of `Socket`. This book is about TCP/IP, so for simplicity we assume that the underlying implementation for all of the these networking classes is the default.

2.2.2 TCP Server

We now turn our attention to constructing a server. The server's job is to set up a communication endpoint and passively wait for connections from clients. The typical TCP server goes through two steps:

1. Construct a `ServerSocket` instance, specifying the local port. This socket listens for incoming connections to the specified port.
2. Repeatedly:
   - Call the `accept()` method of `ServerSocket` to get the next incoming client connection. Upon establishment of a new client connection, an instance of `Socket` for the new connection is created and returned by `accept()`.
   - Communicate with the client using the returned `Socket`'s `InputStream` and `OutputStream`.
   - Close the new client socket connection using the `close()` method of `Socket`.

Our next example, `TCPEchoServer.java`, implements the echo service used by our client program. The server is very simple. It runs forever, repeatedly accepting a connection, receiving and echoing bytes until the connection is closed by the client, and then closing the client socket.

TCPEchoServer.java

```
 0  import java.net.*;  // for Socket, ServerSocket, and InetAddress
 1  import java.io.*;   // for IOException and Input/OutputStream
 2
 3  public class TCPEchoServer {
 4
 5    private static final int BUFSIZE = 32;   // Size of receive buffer
 6
 7    public static void main(String[] args) throws IOException {
 8
 9      if (args.length != 1)  // Test for correct # of args
10        throw new IllegalArgumentException("Parameter(s): <Port>");
11
12      int servPort = Integer.parseInt(args[0]);
13
14      // Create a server socket to accept client connection requests
15      ServerSocket servSock = new ServerSocket(servPort);
16
17      int recvMsgSize;   // Size of received message
18      byte[] byteBuffer = new byte[BUFSIZE];  // Receive buffer
```

```
19
20      for (;;) { // Run forever, accepting and servicing connections
21        Socket clntSock = servSock.accept();     // Get client connection
22
23        System.out.println("Handling client at " +
24          clntSock.getInetAddress().getHostAddress() + " on port " +
25               clntSock.getPort());
26
27        InputStream in = clntSock.getInputStream();
28        OutputStream out = clntSock.getOutputStream();
29
30        // Receive until client closes connection, indicated by -1 return
31        while ((recvMsgSize = in.read(byteBuffer)) != -1)
32          out.write(byteBuffer, 0, recvMsgSize);
33
34        clntSock.close();  // Close the socket.  We are done with this client!
35      }
36      /* NOT REACHED */
37    }
38  }
```

TCPEchoServer.java

1. **Application setup and parameter parsing:** lines 0–12
2. **Server socket creation:** line 15
   servSock listens for client connection requests on the port specified in the constructor.
3. **Loop forever, iteratively handling incoming connections:** lines 20–35
   - **Accept an incoming connection:** line 21
     The sole purpose of a `ServerSocket` instance is to supply a new, connected `Socket` instance for each new TCP connection. When the server is ready to handle a client, it calls `accept()`, which blocks until an incoming connection is made to the `ServerSocket`'s port. `accept()` then returns an instance of `Socket` that is already *connected* to the remote socket and ready for reading and writing.
   - **Report connected client:** lines 23–25
     We can query the newly created `Socket` instance for the address and port of the connecting client. The `getInetAddress()` method of `Socket` returns an instance of `InetAddress` containing the address of the client. We call `getHostAddress()` to return the IP address as a dotted-quad `String`. The `getPort()` method of `Socket` returns the port of the client.
   - **Get socket input and output streams:** lines 27–28
     Bytes written to this socket's `OutputStream` will be read from the client's socket's `InputStream`, and bytes written to the client's `OutputStream` will be read from this socket's `InputStream`.

- **Receive and repeat data until the client closes:** lines 30–32
  The while loop repeatedly reads bytes (when available) from the input stream and immediately writes the same bytes back to the output stream until the client closes the connection. The `read()` method of `InputStream` reads *up to* the maximum number of bytes the array can hold (in this case, BUFSIZE bytes) into the byte array (*byteBuffer*) and returns the number of bytes read. `read()` blocks until data is available and returns −1 if there is no data available, indicating that the client closed its socket. In the echo protocol, the client closes the connection when it has received the number of bytes back that it sent, so in the server we expect to receive a −1 from `read()`. Recall that in the client, receiving a −1 from `read()` indicates an error because it indicates that the server prematurely closed the connection.

  As previously mentioned, `read()` does not have to fill the entire byte array to return. In fact, it can return after having read only a single byte. The `write()` method of `OutputStream` writes *recvMsgSize* bytes from *byteBuffer* to the socket. The second parameter indicates the offset into the byte array of the first byte to send. In this case, 0 indicates to take bytes starting from the front of *byteBuffer*. If we had used the form of `write()` that takes only the buffer argument, *all* the bytes in the buffer array would have been transmitted, possibly including bytes that were not received from the client!
- **Close client socket:** line 34

ServerSocket

Constructors

`ServerSocket`(**int** *localPort*)

`ServerSocket`(**int** *localPort*, **int** *queueLimit*)

`ServerSocket`(**int** *localPort*, **int** *queueLimit*, **InetAddress** *localAddr*)

Construct a TCP socket that is ready to accept incoming connections to the specified local port. Optionally, the size of the connection queue and the local address can be set.

localPort	Local port. A port of 0 allows the constructor to pick any available port.
queueLimit	The maximum size of the queue of incomplete connections and sockets waiting to be `accept()`ed. If a client connection request arrives when the queue is full, the connection is refused. Note that this may not necessarily be a hard limit. For most platforms, it cannot be used to precisely control client population.

localAddr The IP address to which connections to this socket should be addressed (must be one of the local interface addresses). If the address is not specified, the socket will accept connections to any of the host's IP addresses. This may be useful for hosts with multiple interfaces where the server socket should only accept connections on one of its interfaces.

Operators

Socket `accept()`

Returns a connected `Socket` instance for the next new incoming connection to the server socket. If no established connection is waiting, `accept()` blocks until one is established or a timeout occurs (see `setSoTimeout()`).

void `close()`

Closes the underlying TCP socket. After invoking this method, incoming client connection requests for this socket are rejected.

Accessors/Mutators

InetAddress `getInetAddress()`

int `getLocalPort()`

Returns the local address/port of the server socket.

int `getSoTimeout()`

void `setSoTimeout`(**int** *timeout*)

Returns/sets the maximum amount of time (in milliseconds) that an `accept()` will block for this socket. If the timer expires before a connection request arrives, an `InterruptedIOException` is thrown. A timeout value of 0 indicates no timeout: calls to `accept()` will not return until a new connection is available, regardless of how much time passes (see Section 4.2).

2.2.3 Input and Output Streams

As illustrated by the examples above, the primary paradigm for I/O in Java is the *stream* abstraction. A stream is simply an ordered sequence of bytes. Java *input streams* support reading bytes, and *output streams* support writing bytes. In our TCP client and server, each `Socket` instance holds an `InputStream` and an `OutputStream` instance. When we write to the output stream of a `Socket`, the bytes can (eventually) be read from the input stream of the `Socket` at the other end of the connection.

`OutputStream` is the abstract superclass of all output streams in Java. Using an `OutputStream`, we can write bytes to, flush, and close the output stream.

OutputStream

abstract void `write`(**int** *data*)

Writes a single byte to the output stream.

data Byte (low-order 8 bits) to write to output stream

void `write`(**byte[ ]** *data*)

Writes entire array of bytes to the output stream.

data Bytes to write to output stream

void `write`(**byte[ ]** *data*, **int** *offset*, **int** *length*)

Writes *length* bytes from *data* starting from byte *offset*.

data Bytes from which to write to output stream

offset Starting byte to send in *data*

length Number of bytes to send

void `flush`()

Pushes any buffered data out to the stream.

void `close`()

Terminates the stream.

`InputStream` is the abstract superclass of all input streams. Using an `InputStream`, we can read bytes from and close the input stream.

InputStream

abstract int `read`()

Read and return a single byte from the input stream. The byte read is in the least significant byte of the returned integer. This method returns −1 on end-of-stream.

int `read`(**byte[ ]** *data*)

Reads up to *data.length* bytes (or until the end-of-stream) from the input stream into *data* and returns the number of bytes read. If no data is available, `read()` blocks until at least 1 byte can be read or the end-of-stream is detected, indicated by a return of −1.

data Buffer to receive data from input stream

int `read`(**byte[ ]** *data*, **int** *offset*, **int** *length*)

Reads up to *length* bytes (or until the end-of-stream) from the input stream into *data*, starting at position *offset*, and returns the number of bytes read. If no data

is available, `read()` blocks until at least 1 byte can be read or the end-of-stream is detected, indicated by a return of −1.

data	Buffer to receive data from input stream
offset	Starting byte of *data* in which to write
length	Maximum number of bytes to read

int `available()`

Returns the number of bytes available for input.

void `close()`

Terminates the stream.

2.3 UDP Sockets

UDP provides an end-to-end service different from that of TCP. In fact, UDP performs only two functions: 1) it adds another layer of addressing (ports) to that of IP, and 2) it detects data corruption that may occur in transit and discards any corrupted messages. Because of this simplicity, UDP sockets have some different characteristics from the TCP sockets we saw earlier. For example, UDP sockets do not have to be connected before being used. Where TCP is analogous to telephone communication, UDP is analogous to communicating by mail: you do not have to "connect" before you send a package or letter, but you do have to specify the destination address for each one. Similarly, each message—called a *datagram*—carries its own address information and is independent of all others. In receiving, a UDP socket is like a mailbox into which letters or packages from many different sources can be placed. As soon as it is created, a UDP socket can be used to send/receive messages to/from any address and to/from many different addresses in succession.

Another difference between UDP sockets and TCP sockets is the way that they deal with message boundaries: *UDP sockets preserve them*. This makes receiving an application message simpler, in some ways, than it is with TCP sockets. (This is discussed further in Section 2.3.4.) A final difference is that the end-to-end transport service UDP provides is best-effort: there is no guarantee that a message sent via a UDP socket will arrive at its destination, and messages can be delivered in a different order than they were sent (just like letters sent through the mail). A program using UDP sockets must therefore be prepared to deal with loss and reordering. (We'll provide an example of this later.)

Given this additional burden, why would an application use UDP instead of TCP? One reason is efficiency: if the application exchanges only a small amount of data—say, a single request message from client to server and a single response message in the other direction—TCP's connection establishment phase at least doubles the number of messages (and the number of round-trip delays) required for the communication. Another reason is flexibility: when something other than a reliable byte-stream service is required, UDP provides a minimal-overhead platform on which to implement whatever is needed.

Java programmers use UDP sockets via the classes `DatagramPacket` and `DatagramSocket`. Both clients and servers use `DatagramSockets` to send and receive `DatagramPackets`.

2.3.1 DatagramPacket

Instead of sending and receiving streams of bytes as with TCP, UDP endpoints exchange self-contained messages, called datagrams, which are represented in Java as instances of `DatagramPacket`. To send, a Java program constructs a `DatagramPacket` instance and passes it as an argument to the `send()` method of a `DatagramSocket`. To receive, a Java program constructs a `DatagramPacket` instance with preallocated space (a **byte[ ]**), into which the contents of a received message can be copied (if/when one arrives), and then passes the instance to the `receive()` method of a `DatagramSocket`.

In addition to the data, each instance of `DatagramPacket` also contains address and port information, the semantics of which depend on whether the datagram is being sent or received. When a `DatagramPacket` is sent, the address and port identify the destination; for a received `DatagramPacket`, they identify the source of the received message. Thus, a server can receive into a `DatagramPacket` instance, modify its buffer contents, then send the same instance, and the modified message will go back to its origin. Internally, a `DatagramPacket` also has *length* and *offset* fields, which describe the location and number of bytes of message data inside the associated buffer. See the following reference and Section 2.3.4 for some pitfalls to avoid when using `DatagramPackets`.

DatagramPacket

Constructors

`DatagramPacket`(**byte[ ]** *buffer*, **int** *length*)

`DatagramPacket`(**byte[ ]** *buffer*, **int** *offset*, **int** *length*)

`DatagramPacket`(**byte[ ]** *buffer*, **int** *length*, **InetAddress** *remoteAddr*, **int** *remotePort*)

`DatagramPacket`(**byte[ ]** *buffer*, **int** *offset*, **int** *length*, **InetAddress** *remoteAddr*, **int** *remotePort*)

Constructs a datagram and makes the given byte array its data buffer. The first two forms are typically used to construct `DatagramPackets` for receiving because the destination address is not specified (although it could be specified later with `setAddress()` and `setPort()`). The second two forms are typically used to construct `DatagramPackets` for sending.

buffer	Datagram payload
length	Number of bytes of the buffer that will actually be used. If the datagram is sent, *length* bytes will be transmitted. If receiving into this datagram, *length* specifies the maximum number of bytes to be placed in the buffer.
offset	Location in the buffer array of the first byte of message data to be sent/received; defaults to 0 if unspecified.

remoteAddr	Address (typically destination) of the datagram
remotePort	Port (typically destination) of the datagram

Accessors/Mutators

InetAddress `getAddress`()

void `setAddress`(**InetAddress** *address*)

Returns/sets the datagram address. There are other ways to set the address: 1) the address of a `DatagramPacket` instance can also be set by the constructor, and 2) the `receive()` method of `DatagramSocket` sets the address to the datagram sender's address.

address	Datagram address

int `getPort`()

void `setPort`(**int** *port*)

Returns/sets the datagram port. There are other ways to set the address: 1) the port can be explicitly set by the constructor or the `setPort()` method, and 2) the `receive()` method of `DatagramSocket` sets the port to the datagram sender's port.

port	Datagram port

int `getLength`()

void `setLength`(**int** *length*)

Returns/sets the internal length of the datagram. The internal datagram length can be set explicitly by the constructor or by the `setLength()` method. Attempting to make it larger than the length of the associated buffer results in an `IllegalArgumentException`. The `receive()` method of `DatagramSocket` uses the internal length in two ways: 1) on input, it specifies the maximum number of bytes of a received message that will be copied into the buffer, and 2) on return, it indicates the number of bytes actually placed in the buffer.

length	Length in bytes of the usable portion of the buffer

int `getOffset`()

Returns the location in the buffer of the first byte of data to be sent/received. There is no `setOffset()` method; however, it can be set with `setData()`.

byte[ ] `getData`()

Returns the buffer associated with the datagram. The returned object is a reference to the byte array that was most recently associated with this `DatagramPacket`, either by the constructor or by `setData()`. The length of the returned buffer may be greater than the internal datagram length, so the internal length and offset values should be used to determine the actual received data.

void `setData`(**byte[ ]** *buffer*)
void `setData`(**byte[ ]** *buffer*, **int** *offset*, **int** *length*)

Makes the given byte array the datagram buffer. The first form makes the entire byte array the buffer; the second form makes bytes *offset* through *offset* + *length* − 1 the buffer. The first form never updates the internal offset and only updates the internal length if the given buffer's length is less than the current internal length. The second form always updates the internal offset and length.

buffer	Preallocated byte array for datagram packet data
offset	Location in *buffer* where first byte is to be accessed
length	Number of bytes to be read from/written into *buffer*

2.3.2 UDP Client

A UDP client begins by sending a datagram to a server that is passively waiting to be contacted. The typical UDP client goes through three steps:

1. Construct an instance of `DatagramSocket`, optionally specifying the local address and port.
2. Communicate by sending and receiving instances of `DatagramPacket` using the `send()` and `receive()` methods of `DatagramSocket`.
3. When finished, deallocate the socket using the `close()` method of `DatagramSocket`.

Unlike a `Socket`, a `DatagramSocket` is not constructed with a specific destination address. This illustrates one of the major differences between TCP and UDP. A TCP socket is required to establish a connection with another TCP socket on a specific host and port before any data can be exchanged, and, thereafter, it *only* communicates with that socket until it is closed. A UDP socket, on the other hand, is not required to establish a connection before communication, and each datagram can be sent to or received from a different destination. (The `connect()` method of `DatagramSocket` does allow the specification of the remote address and port, but its use is optional.)

Our UDP echo client, `UDPEchoClientTimeout.java`, sends a datagram containing the string to be echoed and prints whatever it receives back from the server. A UDP echo server simply repeats each datagram that it receives back to the client. Of course, a UDP client only communicates with a UDP server. Many systems include a UDP echo server for debugging and testing purposes.

One consequence of using UDP is that datagrams can be lost. In the case of our echo protocol, either the echo request from the client or the echo reply from the server may be lost in the network. Recall that our TCP echo client sends an echo string and then blocks on `read()` waiting for a reply. If we try the same strategy with our UDP echo client and the echo request datagram is lost, our client will block forever on `receive()`. To avoid this problem, our client specifies a maximum amount of time to block on `receive()`, after which it tries again by resending the echo request datagram. Our echo client performs the following steps:

1. Send the echo string to the server.
2. Block on `receive()` for up to three seconds, starting over (up to five times) if the reply is not received before the timeout.
3. Terminate the client.

UDPEchoClientTimeout.java

```
 0  import java.net.*;  // for DatagramSocket, DatagramPacket, and InetAddress
 1  import java.io.*;   // for IOException
 2
 3  public class UDPEchoClientTimeout {
 4
 5    private static final int TIMEOUT = 3000;   // Resend timeout (milliseconds)
 6    private static final int MAXTRIES = 5;     // Maximum retransmissions
 7
 8    public static void main(String[] args) throws IOException {
 9
10      if ((args.length < 2) || (args.length > 3))  // Test for correct # of args
11        throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");
12
13      InetAddress serverAddress = InetAddress.getByName(args[0]);  // Server address
14      // Convert the argument String to bytes using the default encoding
15      byte[] bytesToSend = args[1].getBytes();
16
17      int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
18
19      DatagramSocket socket = new DatagramSocket();
20
21      socket.setSoTimeout(TIMEOUT);  // Maximum receive blocking time (milliseconds)
22
23      DatagramPacket sendPacket = new DatagramPacket(bytesToSend,  // Sending packet
24          bytesToSend.length, serverAddress, servPort);
25
26      DatagramPacket receivePacket =                               // Receiving packet
27          new DatagramPacket(new byte[bytesToSend.length], bytesToSend.length);
28
29      int tries = 0;      // Packets may be lost, so we have to keep trying
30      boolean receivedResponse = false;
31      do {
32        socket.send(sendPacket);          // Send the echo string
33        try {
34          socket.receive(receivePacket);  // Attempt echo reply reception
35
36          if (!receivePacket.getAddress().equals(serverAddress))  // Check source
37            throw new IOException("Received packet from an unknown source");
38
39          receivedResponse = true;
```

```
40        } catch (InterruptedIOException e) {  // We did not get anything
41          tries += 1;
42          System.out.println("Timed out, " + (MAXTRIES - tries) + " more tries...");
43        }
44      } while ((!receivedResponse) && (tries < MAXTRIES));
45
46      if (receivedResponse)
47        System.out.println("Received: " + new String(receivePacket.getData()));
48      else
49        System.out.println("No response -- giving up.");
50
51      socket.close();
52    }
53  }
```

UDPEchoClientTimeout.java

1. **Application setup and parameter parsing:** lines 0–17
   Convert argument to bytes: line 15
2. **UDP socket creation:** line 19
   This instance of `DatagramSocket` can send datagrams to any UDP socket. We do not specify a local address or port so some local address and available port will be selected. We can explicitly set them with the `setLocalAddress()` and `setLocalPort()` methods or in the constructor.
3. **Set the socket timeout:** line 21
   The timeout for a datagram socket controls the maximum amount of time (milliseconds) a call to `receive()` will block. Here we set the timeout to three seconds. Note that timeouts are not precise: the call may block for more than the specified time (but not less).
4. **Create datagram to send:** lines 23–24
   To create a datagram for sending, we need to specify three things: data, destination address, and destination port. For the destination address, we may identify the echo server either by name or IP address. If we specify a name, it is converted to the actual IP address in the constructor.
5. **Create datagram to receive:** lines 26–27
   To create a datagram for receiving, we only need to specify a byte array to hold the datagram data. The address and port of the datagram source will be filled in by `receive()`.
6. **Send the datagram:** lines 29–44
   Since datagrams may be lost, we must be prepared to retransmit the datagram. We loop sending and attempting a receive of the echo reply up to five times.
   - **Send the datagram:** line 32
     `send()` transmits the datagram to the address and port specified in the datagram.

- **Handle datagram reception:** lines 33–43
  `receive()` blocks until it either receives a datagram or the timer expires. Timer expiration is indicated by an `InterruptedIOException`. If the timer expires, we increment the send attempt count (*tries*) and start over. After the maximum number of tries, the while loop exits without receiving a datagram. If `receive()` succeeds, we set the loop flag *receivedResponse* to `true`, causing the loop to exit. Since packets may come from anywhere, we check the source address of the recieved datagram to verify that it matches the address of the specified echo server.

7. **Print reception results:** lines 46–49
   If we received a datagram, *receivedResponse* is `true`, and we can print the datagram data.
8. **Close the socket:** line 51

We invoke the UDP client using the same parameters as used in the TCP client.

DatagramSocket

Constructors

`DatagramSocket()`

`DatagramSocket(`**int** *localPort*`)`

`DatagramSocket(`**int** *localPort*, **InetAddress** *localAddr*`)`

Constructs a UDP socket. Either or both the local port and address may be specified. If the local port is not specified, the socket is bound to any available local port. If the local address is not specified, one of the local addresses is chosen.

localPort	Local port; a localPort of 0 allows the constructor to pick any available port.
localAddr	Local address

Operators

void `close()`

After closing, datagrams may no longer be sent or received using this socket.

void `connect(`**InetAddress** *remoteAddr*, **int** *remotePort*`)`

Sets the remote address and port of the socket. Attempting to send datagrams with a different address will cause an exception to be thrown. The socket will only receive datagrams from the specified port and address. Datagrams from any other port or address are ignored. This is strictly a local operation because there is no end-to-end connection. *Caveat:* A socket that is connected to a multicast or broadcast address can

only send datagrams, because a datagram source address is always a unicast address (see Section 4.3).

remoteAddr Remote address

remotePort Remote port

void `disconnect`()

Removes the remote address and port specification of the socket (see `connect()`).

void `receive`(**DatagramPacket** *packet*)

Places data from the next received message into the given `DatagramPacket`.

packet Receptacle for received information, including source address and port as well as message data. (See the `DatagramPacket` reference for details of semantics.)

void `send`(**DatagramPacket** *packet*)

Sends a datagram from this socket.

packet Specifies the data to send and the destination address and port. If *packet* does not specify a destination address, the `DatagramSocket` must be "connected" to a remote address and port (see `connect()`).

Accessors/Mutators

InetAddress `getInetAddress`()

int `getPort`()

Returns the remote socket address/port.

InetAddress `getLocalAddress`()

int `getLocalPort`()

Returns the local socket address/port.

int `getReceiveBufferSize`()

int `getSendBufferSize`()

void `setReceiveBufferSize`(**int** *size*)

void `setSendBufferSize`(**int** *size*)

The `DatagramSocket` has limits on the maximum datagram size that can be sent/received through this socket. The receive limit also determines the amount of message data that can be queued waiting to be returned via `receive()`. That is, when the amount of buffered data exceeds the limit, arriving packets are quietly discarded. Setting the size is only a hint to the underlying implementation. Also, the semantics of the limit may vary from system to system: it may be a hard limit on some and soft on others.

size Desired limit on packet and/or queue size (bytes)

int `getSoTimeout()`

void `setSoTimeout`(**int** *timeout*)

Returns/sets the maximum amount of time that a `receive()` will block for this socket. If the specified time elapses before data is available, an `InterruptedIOException` is thrown.

timeout	The maximum amount of time (milliseconds) that `receive()` will block for the socket. A timeout of 0 indicates that a receive will block until data is available.

2.3.3 UDP Server

Like a TCP server, a UDP server's job is to set up a communication endpoint and passively wait for the client to initiate the communication; however, since UDP is connectionless, UDP communication is initiated by a datagram from the client, without going through a connection setup as in TCP. The typical UDP server goes through four steps:

1. Construct an instance of `DatagramSocket`, specifying the local port and, optionally, the local address. The server is now ready to receive datagrams from any client.
2. Receive an instance of `DatagramPacket` using the `receive()` method of `DatagramSocket`. When `receive()` returns, the datagram contains the client's address so we know where to send the reply.
3. Communicate by sending and receiving `DatagramPackets` using the `send()` and `receive()` methods of `DatagramSocket`.
4. When finished, deallocate the socket using the `close()` method of `DatagramSocket`.

Our next program example, `UDPEchoServer.java`, implements the UDP version of the echo server. The server is very simple: it loops forever, receiving datagrams and then sending the same datagrams back to the client. Actually, our server only receives and sends back the first 255 (ECHOMAX) characters of the datagram; any excess is silently discarded by the socket implementation (see Section 2.3.4).

UDPEchoServer.java

```
0  import java.net.*;  // for DatagramSocket, DatagramPacket, and InetAddress
1  import java.io.*;   // for IOException
2
3  public class UDPEchoServer {
4
5    private static final int ECHOMAX = 255;  // Maximum size of echo datagram
```

```
 6
 7    public static void main(String[] args) throws IOException {
 8
 9      if (args.length != 1)  // Test for correct argument list
10        throw new IllegalArgumentException("Parameter(s): <Port>");
11
12      int servPort = Integer.parseInt(args[0]);
13
14      DatagramSocket socket = new DatagramSocket(servPort);
15      DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX], ECHOMAX);
16
17      for (;;) {  // Run forever, receiving and echoing datagrams
18        socket.receive(packet);     // Receive packet from client
19        System.out.println("Handling client at " +
20          packet.getAddress().getHostAddress() + " on port " + packet.getPort());
21        socket.send(packet);       // Send the same packet back to client
22        packet.setLength(ECHOMAX); // Reset length to avoid shrinking buffer
23      }
24      /* NOT REACHED */
25    }
26  }
```

UDPEchoServer.java

1. **Application setup and parameter parsing:** lines 0–12
   `UDPEchoServer` takes a single parameter, the local port of the echo server socket.
2. **Create and set up datagram socket:** line 14
   Unlike our UDP client, a UDP server must explicitly set its local port to a number known by the client; otherwise, the client will not know the destination port for its echo request datagram. When the server receives the echo datagram from the client, it can find out the client's address and port from the datagram.
3. **Create datagram:** line 15
   UDP messages are contained in datagrams. We construct an instance of `DatagramPacket` with a buffer of ECHOMAX (255) bytes. This datagram will be used both to receive the echo request and to send the echo reply.
4. **Iteratively handle incoming echo requests:** lines 17–23
   The UDP server uses a single socket for all communication, unlike the TCP server, which creates a new socket with every successful `accept()`.
   - **Receive an echo request datagram:** lines 18–20
     The `receive()` method of `DatagramSocket` blocks until a datagram is received from a client (unless a timeout is set). There is no connection, so each datagram may come

from a different sender. The datagram itself contains the sender's (client's) source address and port.

- **Send echo reply:** line 21
  packet already contains the echo string and echo reply destination address and port, so the `send()` method of `DatagramSocket` can simply transmit the datagram previously received. Note that when we receive the datagram, we interpret the datagram address and port as the *source* address and port, and when we send a datagram, we interpret the datagram's address and port as the *destination* address and port.
- **Reset buffer size:** line 22
  The internal length of *packet* was set to the length of the message just processed, which may have been smaller than the original buffer size. If we do not reset the internal length before receiving again, the next message will be truncated if it is longer than the one just received.

2.3.4 Sending and Receiving with UDP Sockets

A subtle but important difference between TCP and UDP is that UDP preserves message boundaries. Each call to `receive()` returns data from at most one call to `send()` Moreover, different calls to `receive()` will never return data from the same call to `send()`.

When a call to `write()` on a TCP socket's output stream returns, all the caller knows is that the data has been copied into a buffer for transmission; the data may or may not have actually been transmitted yet. (This is covered in more detail in Chapter 5.) UDP, however, does not provide recovery from network errors and, therefore, does not buffer data for possible retransmission. This means that by the time a call to `send()` returns, the message has been passed to the underlying channel for transmission and is (or soon will be) on its way out the door.

Between the time a message arrives from the network and the time its data is returned via `read()` or `receive()`, the data is stored in a *first-in, first-out (FIFO)* queue of received data. With a connected TCP socket, all received-but-not-yet-delivered bytes are treated as one continuous sequence of bytes (see Chapter 5). For a UDP socket, however, the received data may have come from different senders. A UDP socket's received data is kept in a queue of messages, each with associated information identifying its source. A call to `receive()` will never return more than one message. However, if `receive()` is called with a `DatagramPacket` containing a buffer of size n, and the size of the first message in the receive queue exceeds n, only the first n bytes of the message are returned. The remaining bytes are quietly discarded, with no indication to the receiving program that information has been lost!

For this reason, a receiver should always supply a `DatagramPacket` with a buffer big enough to hold the largest message allowed by its application protocol at the time it calls `receive()`. This technique will guarantee that no data will be lost. The maximum amount of data that can be transmitted in a `DatagramPacket` is 65,507 bytes—the largest payload that can be carried in a UDP datagram. It is important to remember here that each instance of `DatagramPacket` has an internal notion of message length that may be changed whenever a message is received into

that instance (to reflect the number of bytes in the received message). Applications that call `receive()` more than once with the same instance of `DatagramPacket` should explicitly reset the internal length to the actual buffer length before each subsequent call to `receive()`.

Another potential source of problems for beginners is the `getData()` method of `DatagramPacket`, which always returns the entire original buffer, ignoring the internal offset and length values. Receiving a message into the `DatagramPacket` only modifies those locations of the buffer into which message data was placed. For example, suppose *buf* is a byte array of size 20, which has been initialized so that each byte contains its index in the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Suppose also that *dg* is a `DatagramPacket`, and that we set *dg*'s buffer to be the middle 10 bytes of *buf*:

```
dg.setData(buf,5,10);
```

Now suppose that *dgsocket* is a `DatagramSocket`, and that somebody sends an 8-byte message containing

41	42	43	44	45	46	47	48

to *dgsocket*. The message is received into *dg*:

```
dgsocket.receive(dg);
```

Now, calling `dg.getData()` returns a reference to the original byte array *buf*, whose contents are now

0	1	2	3	4	41	42	43	44	45	46	47	48	13	14	15	16	17	18	19

Note that only bytes 5–12 of *buf* have been modified and that, in general, the application needs to use `getOffset()` and `getData()` to access just the received data. One possibility is to copy the received data into a separate byte array, like this:

```
byte[] destBuf = new byte[dg.getLength()];
System.arraycopy(dg.getData(), dg.getOffset(), destBuf, 0, destBuf.length);
```

2.4 Exercises

1. For `TCPEchoServer.java`, we explicitly specify the port to the socket in the constructor. We said that a socket must have a port for communication, yet we do not specify a port in `TCPEchoClient.java`. How is the echo client's socket assigned a port?
2. When you make a phone call, it is usually the callee that answers with "Hello." What changes to our client and server examples would be needed to implement this?
3. What happens if a TCP server never calls `accept()`? What happens if a TCP client sends data on a socket that has not yet been `accept()`ed at the server?

4. Servers are supposed to run for a long time without stopping—therefore, they must be designed to provide good service no matter what their clients do. Examine the server examples (`TCPEchoServer.java` and `UDPEchoServer.java`) and list anything you can think of that a client might do to cause it to give poor service to other clients. Suggest improvements to fix the problems that you find.
5. Modify `TCPEchoServer.java` to read and write only a single byte at a time, sleeping one second between each byte. Verify that `TCPEchoClient.java` requires multiple reads to successfully receive the entire echo string, even though it sent the echo string with one `write()`.
6. Modify `TCPEchoServer.java` to read and write a single byte and then close the socket. What happens when the `TCPEchoClient` sends a multibyte string to this server? What is happening? (Note that the response could vary by OS.)
7. Modify `UDPEchoServer.java` so that it only echoes every other datagram it receives. Verify that `UDPEchoClientTimeout.java` retransmits datagrams until it either receives a reply or exceeds the number of retries.
8. Modify `UDPEchoServer.java` so that ECHOMAX is much shorter (say, 5 bytes). Then use `UDPEchoClientTimeout.java` to send an echo string that is too long. What happens?
9. Verify experimentally the size of the largest message you can send and receive using a `DatagramPacket`.
10. While `UDPEchoServer.java` explicitly specifies its local port in the constructor, we do not specify the local port in `UDPEchoClientTimeout.java`. How is the UDP echo client's socket given a port number? *Hint:* The answer is different for TCP.

This Page Intentionally Left Blank

chapter 3

Sending and Receiving Messages

When writing programs to communicate via sockets, you will generally be implementing an *application protocol* of some sort. Typically you use sockets because your program needs to provide information to, or use information provided by, another program. There is no magic: sender and receiver must agree on how this information will be encoded, who sends what information when, and how the communication will be terminated. In our echo example, the application protocol is trivial: neither the client's nor the server's behavior is affected by the *contents* of the bytes they exchange. Because most applications require that the behaviors of client and server depend upon the *information* they exchange, application protocols are usually more complicated.

The TCP/IP protocols transport bytes of user data without examining or modifying them. This allows applications great flexibility in how they encode their information for transmission. For various reasons, most application protocols are defined in terms of discrete *messages* made up of sequences of *fields*. Each field contains a specific piece of information encoded as a sequence of bits. The application protocol specifies exactly how these sequences of bits are to be formatted by the sender and interpreted, or *parsed*, by the receiver so that the latter can extract the meaning of each field. About the only constraint imposed by TCP/IP is that information must be sent and received in chunks whose length in bits is a multiple of eight. So from now on we consider messages to be sequences of *bytes*. Given this, it may be helpful to think of a transmitted message as a sequence of numbers, each between 0 and 255, inclusive (that being the range of binary values that can be encoded in 8 bits—1 byte).

As a concrete example for this chapter, let's consider the problem of transferring price quote information between vendors and buyers. A simple quote for some quantity of a particular item might include the following information:

Item number: A large integer identifying the item

Item description: A text string describing the item

Unit price: The cost per item in cents

Quantity: The number of units offered at that price

Discounted?: Whether the price includes a discount

In stock?: Whether the item is in stock

We collect this information in a class `ItemQuote.java`. For convenience in viewing the information in our program examples, we include a `toString()` method. Throughout this chapter, the variable *item* refers to an instance of `ItemQuote`.

ItemQuote.java

```
 0  public class ItemQuote {
 1
 2    public long itemNumber;        // Item identification number
 3    public String itemDescription; // String description of item
 4    public int quantity;           // Number of items in quote (always >= 1)
 5    public int unitPrice;          // Price (in cents) per item
 6    public boolean discounted;     // Price reflect a discount?
 7    public boolean inStock;        // Item(s) ready to ship?
 8
 9    public ItemQuote(long itemNumber, String itemDescription,
10        int quantity, int unitPrice, boolean discounted, boolean inStock)  {
11      this.itemNumber = itemNumber;
12      this.itemDescription = itemDescription;
13      this.quantity = quantity;
14      this.unitPrice = unitPrice;
15      this.discounted = discounted;
16      this.inStock = inStock;
17    }
18
19    public String toString() {
20      final String EOLN = java.lang.System.getProperty("line.separator");
21      String value = "Item#=" + itemNumber + EOLN +
22                     "Description=" + itemDescription + EOLN +
23                     "Quantity=" + quantity + EOLN +
24                     "Price(each)=" + unitPrice + EOLN +
25                     "Total=" + (quantity * unitPrice);
26      if (discounted)
27        value += " (discounted)";
28      if (inStock)
29        value += EOLN + "In Stock" + EOLN;
30      else
31        value += EOLN + "Out of Stock" + EOLN;
32      return value;
33    }
34  }
```

3.1 Encoding Information

What if a client program needs to obtain quote information from a vendor program? The two programs must agree on how the information contained in the `ItemQuote` will be represented as a sequence of bytes "on the wire"—sent over a TCP connection or carried in a UDP datagram. (Note that everything in this chapter also applies if the "wire" is a file that is written by one program and read by another.) In our example, the information to be represented consists of primitive types (integers, booleans) and a character string.

Transmitting information via the network in Java requires that it be written to an `OutputStream` (of a `Socket`) or encapsulated in a `DatagramPacket` (which is then sent via a `DatagramSocket`). However, the only data types to which these operations can be applied are **byte**s and arrays of **byte**s. As a strongly typed language, Java requires that other types—`String`, **int**, and so on—be explicitly converted to these transmittable types. Fortunately, the language has a number of built-in facilities that make such conversions more convenient. Before dealing with the specifics of our example, however, we focus on some basic concepts of representing information as sequences of bytes for transmission.

3.1.1 Text

Old-fashioned text—strings of printable (displayable) characters—is perhaps the most common form of information representation. When the information to be transmitted is natural language, text is the most natural representation. Text is convenient for other forms of information because humans can easily deal with it when printed or displayed; numbers, for example, can be represented as strings of decimal digits.

To send text, the string of characters is translated into a sequence of bytes according to a *character set*. The canonical example of a character encoding system is the venerable *American Standard Code for Information Interchange* (ASCII), which defines a one-to-one mapping between a set of the most commonly used printable characters in English, and binary values. For example, in ASCII the digit `0` is represented by the byte value 48, `1` by 49, and so on up to `9`, which is represented by the byte value 57. ASCII is adequate for applications that only need to exchange English text. As the economy becomes increasingly globalized, however, applications need to deal with other languages, including many that use characters for which ASCII has no encoding, and even some (e.g., Chinese) that use more than 256 characters and thus require more than 1 byte per character to encode. Encodings for the world's languages are defined by companies and by standards bodies. *Unicode* is the most widely recognized such character encoding; it is standardized by the International Organization for Standardization (ISO).

Fortunately, Java provides good support for internationalization, in several ways. First, Java uses Unicode to represent characters internally. Unicode defines a 16-bit (2-byte) code for each character and thus supports a much larger set of characters than ASCII. In fact, the Unicode standard currently defines codes for over 49,000 characters and covers "the principal written languages and symbol systems of the world." [21] Second, Java supports various other standard encodings and provides a clean separation between its internal representation and the encoding used when characters are input or output.

The `getBytes()` methods of class `String` implement this internal-to-external conversion, returning the sequence of bytes that represent the given string in some *external* encoding—either the default encoding or an explicitly named one. (This type of conversion may also happen implicitly—for example, by writing a string to an instance of `OutputStreamWriter`.) Similarly, `String` provides constructors that take a byte array and the name of a particular encoding, and return a `String` instance containing the sequence of characters represented by the byte sequence according to the encoding. (If no encoding is explicitly requested, the default encoding for the platform is used.)

Suppose the value of *item.itemNumber* is 123456. Using ASCII, that part of the string representation of *item* produced by `toString()` would be encoded as

105	116	101	109	35	61	49	50	51	52	53	54
'i'	't'	'e'	'm'	'#'	'='	'1'	'2'	'3'	'4'	'5'	'6'

Using the "ISO8859_1" encoding would produce the same sequence of byte values, because the *International Standard 8859-1* encoding (which is also known as *ISO Latin 1*) is an extension of ASCII—it maps the characters of the ASCII set to the same values as ASCII. However, if we used the North American version of IBM's *Extended Binary Coded Decimal Interchange Code* (EBCDIC), known in Java as the encoding "Cp037," the result would be rather different:

137	163	133	148	123	126	241	242	243	244	245	246
'i'	't'	'e'	'm'	'#'	'='	'1'	'2'	'3'	'4'	'5'	'6'

If we used Unicode, the result would use 2 bytes per character, with 1 byte containing zero and the other byte containing the same value as with ASCII. Obviously the primary requirement in dealing with character encodings is that the sender and receiver must agree on the code to be used.

3.1.2 Binary Numbers

Transmitting large numbers as text strings is not very efficient: each character in the digit string has one of only 10 values, which can be represented using, on average, less than 4 bits per digit. Yet the standard character codes invariably use at least 8 bits per character. Moreover, it is inconvenient to perform arithmetic computation and comparisons with numbers encoded as strings. For example, a receiving program computing the total cost of a quote (quantity times unit price) will generally need to convert both amounts to the local computer's native (binary) integer representation before the computation can be performed. For a more compact and computation-friendly encoding, we can transmit the values of the integers in our data as binary values. To send binary integers as byte sequences, the sender and receiver need to agree on several things:

- *Integer size:* How many bits are used to represent the integer? The sizes of Java's integer types are fixed by the language definition—**short**s are 2 bytes, **int**s are 4, **long**s are 8—so a Java sender and receiver only need to agree on the primitive type to be used. (Communicating with a non-Java application may be more complex.) The size of an integer

type, along with the encoding (signed/unsigned, see below), determines the maximum and minimum values that can be represented using that type.

- *Byte order:* Are the bytes of the binary representation written to the stream (or placed in the byte array) from left to right or right to left? If the most significant byte is transmitted first and the least significant byte is transmitted last, that's the so-called big-endian order. Little-endian is, of course, just the opposite.
- *Signed or unsigned:* Signed integers are usually transmitted in *two's-complement* representation. For k-bit numbers, the two's-complement encoding of the negative integer $-n$, $1 \le n \le 2^{k-1}$, is the binary value of $2^k - n$; and the non-negative integer p, $0 \le p \le 2^{k-1} - 1$, is encoded simply by the k-bit binary value of p. Thus, given k bits, two's complement can represent values in the range -2^{k-1} through $2^{k-1} - 1$, and the most significant bit (msb) tells whether the value is positive (msb = 0) or negative (msb = 1). On the other hand, a k-bit *unsigned* integer can encode values in the range 0 through $2^k - 1$ directly.

Consider again the *itemNumber*. It is a **long**, so its binary representation is 64 bits (8 bytes). If its value is 12345654321 and the encoding is big-endian, the 8 bytes sent would be (with the byte on the left transmitted first):

0	0	0	2	223	219	188	49

If, on the other hand, the value was sent in little-endian order, the transmitted byte values would be:

49	188	219	223	2	0	0	0

If the sender uses big-endian when the receiver is expecting little-endian, the receiver will end up with an *itemNumber* of 3583981154337816576! Most network protocols specify big-endian byte order; in fact it is sometimes called *network byte order*.

Note that the most significant bit of the 64-bit binary value of 12345654321 is 0, so its signed (two's-complement) and unsigned representations are the same. More generally, the distinction between k-bit signed and unsigned values is irrelevant for values that lie in the range 0 through $2^{k-1} - 1$. Unfortunately, protocols often use unsigned integers; Java's lack of unsigned integer types means that some care is required in dealing with such values, especially in decoding. (See program `ItemQuoteDecoderBin.java` in Section 3.4.2 for an example.)

As with strings, Java provides mechanisms to turn primitive integer types into sequences of bytes and vice versa. In particular, streams that support the `DataOutput` interface have methods `writeShort()`, `writeInt()`, and `writeLong()`, which allow those types to be written out directly. These methods all write out the bytes of integer primitive types in big-endian byte order using two's-complement representation. Similarly, implementations of the `DataInput` interface have methods `readInt()`, `readShort()`, and so on. The next section describes some ways to compose instances of these classes.

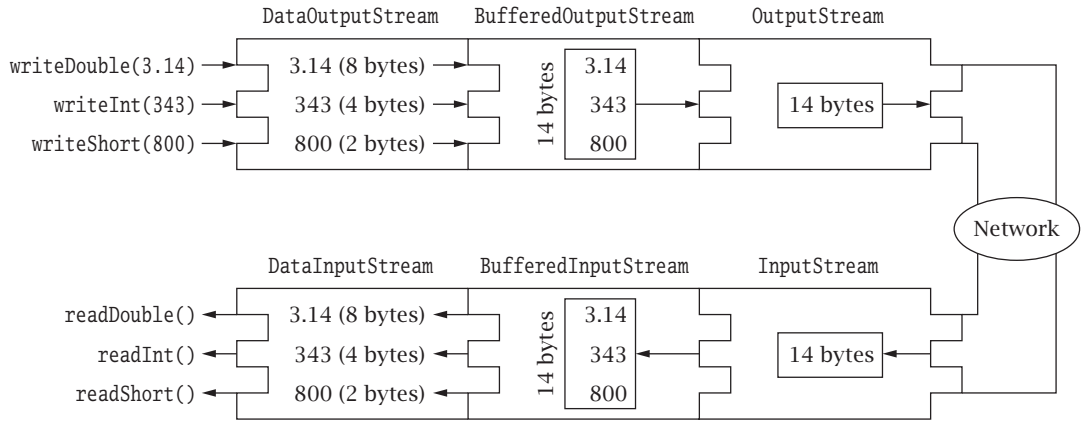


Figure 3.1: Stream composition.

3.2 Composing I/O Streams

Java's stream classes can be composed to provide powerful encoding and decoding facilities. For example, we can wrap the `OutputStream` of a `Socket` instance in a `BufferedOutputStream` instance to improve performance by buffering bytes temporarily and flushing them to the underlying channel all at once. We can then wrap that instance in a `DataOutputStream` to send primitive data types. We would code this composition as follows:

```
Socket socket = new Socket(server, port);
DataOutputStream out = new DataOutputStream(
                          new BufferedOutputStream(socket.getOutputStream()));
```

Figure 3.1 demonstrates this composition. Here, we write our primitive data values, one by one, to `DataOutputStream`, which writes the binary data to `BufferedOutputStream`, which buffers the data from the three writes and then writes once to the socket `OutputStream`, which controls writing to the network. We create a parallel composition for the `InputStream` on the other endpoint to efficiently receive primitive data types.

A complete description of the Java I/O API is beyond the scope of this text; however, Table 3.1 provides a list of some of the relevant Java I/O classes as a starting point for exploiting its capabilities.

3.3 Framing and Parsing

Converting data to wire format is of course only half the story; the original information must be recovered at the receiver from the transmitted sequence of bytes. Application protocols typically deal with discrete messages, which are viewed as collections of fields. *Framing* refers to the problem of enabling the receiver to locate the beginning and end of the message in

I/O Class	Function
Buffered[Input/Output]Stream	Performs buffering for I/O optimization.
Checked[Input/Output]Stream	Maintains a checksum on data.
Data[Input/Output]Stream	Handles read/write for primitive data types.
Digest[Input/Output]Stream	Maintains a digest on data.
GZIP[Input/Output]Stream	De/compresses a byte stream in GZIP format.
Object[Input/Output]Stream	Handles read/write objects and primitive data types.
PushbackInputStream	Allows a byte or bytes to be "unread."
PrintOutputStream	Prints string representation of data type.
ZIP[Input/Output]Stream	De/compresses a byte stream in ZIP format.

Table 3.1: Java I/O Classes

the stream, and of the fields within the message. Whether information is encoded as text, as multibyte binary numbers, or as some combination of the two, the application protocol must enable the receiver of a message to determine when it has received all of the message and to parse it into fields.

If the fields in a message all have fixed sizes and the message is made up of a fixed number of fields, then the size of the message is known in advance and the receiver can simply read the expected number of bytes into a **byte[ ]** buffer. This technique was used in `TCPEchoClient.java`, where we knew the number of bytes to expect from the server. However, when some field (and/or the whole message) can vary in length, as with the *itemDescription* in our example, we do not know beforehand how many bytes to read.

Marking the end of the message is easy in the special case of the last message to be sent on a TCP connection: the sender simply closes the sending side of the connection (using `shutdownOutput()` or `close()`) after sending the message. After the receiver reads the last byte of the message, it receives an end-of-stream indication (i.e., `read()` returns −1), and thus can tell that it has as much of the message as there will ever be. The same principle applies to the last field in a message sent as a `DatagramPacket`.

In all other cases, the message itself must contain additional framing information enabling the receiver to parse the field/message. This information typically takes one of the following forms:

- *Delimiter*: The end of the variable-length field or message is indicated by a *unique marker*, an explicit byte sequence that immediately follows, but does not occur in, the data.
- *Explicit length*: The variable-length field or message is preceded by a (fixed-size) length field that tells how many bytes it contains.

The delimiter-based approach is often used with variable-length text: A particular character or sequence of characters is defined to mark the end of the field. If the entire message consists of text, it is straightforward to read in characters using an instance of `Reader` (which

handles the byte-to-character translation), looking for the delimiter sequence, and returning the character string preceding it.

Unfortunately, the `Reader` classes do not support reading binary data. Moreover, the relationship between the number of *bytes* read from the underlying `InputStream` and the number of *characters* read from the `Reader` is unspecified, especially with multibyte encodings. When a message uses a combination of the two framing methods mentioned above, with some explicit-length-delimited fields and others using character markers, this can create problems.

The class `Framer`, defined below, allows an `InputStream` to be parsed as a sequence of fields delimited by specific byte patterns. The static method `Framer.nextToken()` reads bytes from the given `InputStream` until it encounters the given sequence of bytes or the stream ends. All bytes read up to that point are then returned in a new byte array. If the end of the stream is encountered before any data is read, `null` is returned. The delimiter can be different for each call to `nextToken()`, and the method is completely independent of any encoding.

A couple of words of caution are in order here. First, `nextToken()` is terribly inefficient; for real applications, a more efficient pattern-matching algorithm should be used. Second, when using `Framer.nextToken()` with text-based message formats, the caller must convert the delimiter from a character string to a byte array and the returned byte array to a character string. In this case the character encoding needs to distribute over concatenation, so that it doesn't matter whether a string is converted to bytes all at once, or a little bit at a time.

To make this precise, let $E()$ represent an encoding—that is, a function that maps character sequences to byte sequences. Let a and b be sequences of characters, so $E(a)$ denotes the sequence of bytes that is the result of encoding a. Let "+" denote concatenation of sequences, so $a + b$ is the sequence consisting of a followed by b. This explicit-conversion approach (as opposed to parsing the message as a character stream) should only be used with encodings that have the property that $E(a + b) = E(a) + E(b)$; otherwise, the results may be unexpected. Although most encodings supported in Java have this property, some do not. In particular, UnicodeBig and UnicodeLittle encode a `String` by first outputting a byte-order indicator (the 2-byte sequence 254–255 for big-endian, and 255–254 for little-endian), followed by the 16-bit Unicode value of each character in the `String`, in the indicated byte order. Thus, the encoding of "Big fox" using UnicodeBig is as follows:

254	255	0	66	0	105	0	103	0	32	0	102	0	111	0	120
[mark]		'B'		'i'		'g'		' '		'f'		'o'		'x'	

while the encoding of "Big" concatenated with the encoding of "fox", using the same encoding, is as follows:

254	255	0	66	0	105	0	103	254	255	0	32	0	102	0	111	0	120
[mark]		'B'		'i'		'g'		[mark]		' '		'f'		'o'		'x'	

Using either of these encodings to convert the delimiter results in a byte sequence that begins with the byte-order marker. Moreover, if the byte array returned by `nextToken()` does not begin with one of the markers, any attempt to convert it to a `String` using one of these encodings

will throw an exception. The encodings UnicodeBigUnmarked and UnicodeLittleUnmarked (supported in JDK as of 1.3) omit the byte-order marker, so they are suitable for use with `Framer.nextToken()`.

Framer.java

```
 0  import java.io.*;  // for InputStream and ByteArrayOutputStream
 1
 2  public class Framer {
 3
 4    public static byte[] nextToken(InputStream in, byte[] delimiter)
 5        throws IOException {
 6      int nextByte;
 7
 8      // If the stream has already ended, return null
 9      if ((nextByte = in.read()) == -1)
10        return null;
11
12      ByteArrayOutputStream tokenBuffer = new ByteArrayOutputStream();
13      do {
14        tokenBuffer.write(nextByte);
15        byte[] currentToken = tokenBuffer.toByteArray();
16        if (endsWith(currentToken, delimiter)) {
17          int tokenLength = currentToken.length - delimiter.length;
18          byte[] token = new byte[tokenLength];
19          System.arraycopy(currentToken, 0, token, 0, tokenLength);
20          return token;
21        }
22      } while ((nextByte = in.read()) != -1);   // Stop on end-of-stream
23      return tokenBuffer.toByteArray();         // Received at least 1 byte
24    }
25
26    // Returns true if value ends with the bytes in suffix
27    private static boolean endsWith(byte[] value, byte[] suffix) {
28      if (value.length < suffix.length)
29        return false;
30
31      for (int offset = 1; offset <= suffix.length; offset++)
32        if (value[value.length - offset] != suffix[suffix.length - offset])
33           return false;
34      return true;
35    }
36  }
```

1. `nextToken()`: lines 4–24
   Read from input stream until delimiter or end-of-stream.
   - **Test for end-of-stream:** lines 8–10
     If the input stream is already at end-of-stream, return `null`.
   - **Create a buffer to hold the bytes of the token:** line 12
     We use a `ByteArrayOutputStream` to collect the data byte by byte. The `ByteArray[Input|Output]Stream` classes allow a byte array to be handled like a stream of bytes.
   - **Put the last byte read into the buffer :** line 14
   - **Get a byte array containing the input so far:** line 15
     It is very inefficient to create a new byte array on each iteration, but it is simple.
   - **Check whether the delimiter is a suffix of the current token:** lines 16–21
     If so, create a new byte array containing the bytes read so far, minus the delimiter suffix, and return it.
   - **Get next byte:** line 22
   - **Return the current token on end-of-stream:** line 23
2. `endswith()`: lines 26–35
   - **Compare lengths:** lines 28–29
     The candidate sequence must be at least as long as the delimiter to be a match.
   - **Compare bytes, return `false` on any difference:** lines 31–33
     Compare the last *delim.length* bytes of the token to the delimiter.
   - **If no difference, return `true`:** line 34

3.4 Implementing Wire Formats in Java

To emphasize the fact that the same information can be represented "on the wire" in different ways, we define an interface `ItemQuoteEncoder`, which has a single method that takes an `ItemQuote` instance and converts it to a **byte[ ]** that can be written to an `OutputStream` or encapsulated in a `DatagramPacket`.

ItemQuoteEncoder.java

```
0  public interface ItemQuoteEncoder {
1    byte[] encode(ItemQuote item) throws Exception;
2  }
```

ItemQuoteEncoder.java

The specification of the corresponding decoding functionality is given by the `ItemQuoteDecoder` interface, which has methods for parsing messages received via streams or in `Data-`

gramPackets. Each method performs the same function: extracting the information for one message and returning an ItemQuote instance containing the information.

ItemQuoteDecoder.java

```
0  import java.io.*;   // for InputStream and IOException
1  import java.net.*;  // for DatagramPacket
2
3  public interface ItemQuoteDecoder {
4    ItemQuote decode(InputStream source) throws IOException;
5    ItemQuote decode(DatagramPacket packet) throws IOException;
6  }
```

ItemQuoteDecoder.java

Sections 3.4.1 and 3.4.2 present two different implementations for these interfaces: one using a text representation, the other, a hybrid encoding.

3.4.1 Text-Oriented Representation

Clearly we can represent the ItemQuote information as text. One possibility is to simply transmit the output of the toString() method using a suitable character encoding. To simplify parsing, the approach in this section uses a different representation, in which the values of *itemNumber*, *itemDescription*, and so on are transmitted as a sequence of delimited text fields. The sequence of fields is as follows:

⟨Item Number⟩ ⟨Description⟩ ⟨Quantity⟩ ⟨Price⟩ ⟨Discount?⟩ ⟨In Stock?⟩

The Item Number field (and the other integer-valued fields, Quantity and Price) contain a sequence of decimal digit characters followed by a space character (the delimiter). The Description field is just the description text. However, because the text itself may include the space character, we have to use a different delimiter; we choose the newline character, represented as \n in Java, as the delimiter for this field.

Boolean values can be encoded in several different ways. One possibility is to include the string "true" or the string "false", according to the value of the variable. A more compact approach (and the one used here) is to encode both values (*discounted* and *inStock*) in a single field; the field contains the character 'd' if *discounted* is true, indicating that the item is discounted, and the character 's' if *inStock* is true, indicating that the item is in stock. The absence of a character indicates that the corresponding boolean is false, so this field may be empty. Again, a different delimiter (\n) is used for this final field, to make it slightly easier to recognize the end of the message even when this field is empty. A quote for 23 units of item number 12345, which has the description "AAA Battery" and price $14.45, and which is both in stock and discounted, would be represented as

```
12345 AAA Battery\n23 1445 ds\n
```

Constants needed by both the encoder and the decoder are defined in the `ItemQuoteTextConst` interface, which defines "ISO8859_1" as the default encoding (we could just as easily have used any other encoding as the default) and 1024 as the maximum length (in bytes) of an encoded message. Limiting the length of an encoded message limits the flexibility of the protocol, but it also provides for sanity checks by the receiver.

ItemQuoteTextConst.java

```
0  public interface ItemQuoteTextConst {
1    public static final String DEFAULT_ENCODING = "ISO_8859_1";
2    public static final int MAX_WIRE_LENGTH = 1024;
3  }
```

ItemQuoteTextConst.java

`ItemQuoteEncoderText` implements the text encoding.

ItemQuoteEncoderText.java

```
0  import java.io.*;  // for ByteArrayOutputStream and OutputStreamWriter
1
2  public class ItemQuoteEncoderText implements ItemQuoteEncoder, ItemQuoteTextConst {
3
4    private String encoding;  // Character encoding
5
6    public ItemQuoteEncoderText() {
7      encoding = DEFAULT_ENCODING;
8    }
9
10   public ItemQuoteEncoderText(String encoding) {
11     this.encoding = encoding;
12   }
13
14   public byte[] encode(ItemQuote item) throws Exception {
15     ByteArrayOutputStream buf = new ByteArrayOutputStream();
16     OutputStreamWriter out = new OutputStreamWriter(buf, encoding);
17     out.write(item.itemNumber + " ");
18     if (item.itemDescription.indexOf('\n') != -1)
19       throw new IOException("Invalid description (contains newline)");
20     out.write(item.itemDescription + "\n" + item.quantity + " " +
21               item.unitPrice + " ");
22     if (item.discounted)
23       out.write('d');   // Only include 'd' if discounted
24     if (item.inStock)
25       out.write('s');   // Only include 's' if in stock
```

```
26      out.write('\n');
27      out.flush();
28      if (buf.size() > MAX_WIRE_LENGTH)
29        throw new IOException("Encoded length too long");
30      return buf.toByteArray();
31    }
32  }
```

ItemQuoteEncoderText.java

1. **Constructors:** lines 6–12
   If no encoding is explicitly specified, we use the default encoding specified in the constant interface.
2. `encode()` **method:** lines 14–31
   - **Create an output buffer:** lines 15–16
     A `ByteArrayOutputStream` collects the bytes to be returned. Wrapping it in an `OutputWriter` allows us to take advantage of the latter's methods for converting strings to bytes.
   - **Write the first integer, followed by a space delimiter:** line 17
   - **Check for delimiter:** lines 18–19
     Make sure that the field delimiter is not contained in the field itself. If it is, throw an exception.
   - **Output** *itemDescription* **and other integers:** lines 20–21
   - **Write the flag characters if the booleans are true:** lines 22–25
   - **Write the delimiter for the flag field:** line 26
   - **Flush the output stream:** lines 27–29
     Flush everything to the underlying stream, and call `size()` to check that the resulting byte sequence is not too long. The length restriction allows the receiver to know how big a buffer is needed to receive into a `DatagramPacket`. (For stream communication, this is not necessary, but it is still convenient.)
   - **Return the byte array from the output stream:** line 30

The decoding class `ItemQuoteDecoderText` simply inverts the encoding process.

ItemQuoteDecoderText.java

```
0  import java.io.*;  // for InputStream, ByteArrayInputStream, and IOException
1  import java.net.*; // for DatagramPacket
2
3  public class ItemQuoteDecoderText implements ItemQuoteDecoder, ItemQuoteTextConst {
4
5    private String encoding;  // Character encoding
```

```
 6
 7    public ItemQuoteDecoderText() {
 8      encoding = DEFAULT_ENCODING;
 9    }
10
11    public ItemQuoteDecoderText(String encoding) {
12      this.encoding = encoding;
13    }
14
15    public ItemQuote decode(InputStream wire) throws IOException {
16      String itemNo, description, quant, price, flags;
17      byte[] space = " ".getBytes(encoding);
18      byte[] newline = "\n".getBytes(encoding);
19      itemNo = new String(Framer.nextToken(wire, space), encoding);
20      description = new String(Framer.nextToken(wire, newline), encoding);
21      quant = new String(Framer.nextToken(wire, space), encoding);
22      price = new String(Framer.nextToken(wire, space), encoding);
23      flags = new String(Framer.nextToken(wire, newline), encoding);
24      return new ItemQuote(Long.parseLong(itemNo), description,
25                           Integer.parseInt(quant),
26                           Integer.parseInt(price),
27                           (flags.indexOf('d') != -1),
28                           (flags.indexOf('s') != -1));
29    }
30
31    public ItemQuote decode(DatagramPacket p) throws IOException {
32      ByteArrayInputStream payload =
33        new ByteArrayInputStream(p.getData(), p.getOffset(), p.getLength());
34      return decode(payload);
35    }
36  }
```

ItemQuoteDecoderText.java

1. **Variables and constructors:** lines 5–13
   - **Encoding:** line 5
     The encoding used in the decoder must be the same as in the encoder!
   - **Constructors:** lines 7–13
     If no encoding is given at construction time, the default defined in `ItemQuoteDecoderTextConst` is used.
2. **Stream** `decode()`**:** lines 15–29
   - **Convert delimiters:** lines 17–18
     We get the encoded form of the delimiters ahead of time, for efficiency.

- **Call the** `nextToken()` **method for each field:** lines 19–23
  For each field, we call `Framer.nextToken()` with the appropriate delimiter and convert the result according to the specified encoding.
- **Construct** `ItemQuote`**:** lines 24–28
  Convert to native types using the wrapper conversion methods and test for the presence of the flag characters in the last field.

3. **Packet** `decode()`**:** lines 31–35
   Extract the data, convert to a stream, and call the stream `decode()` method.

3.4.2 Combined Data Representation

Our next encoding represents the integers of the `ItemQuote` as fixed-size, binary numbers: *itemNumber* as 64 bits, and *quantity* and *unitPrice* as 32 bits. It encodes the boolean values as flag bits, which occupy the smallest possible space in an encoded message. Also, the variable-length string *itemDescription* is encoded in a field with an explicit length indication. The binary encoding and decoding share coding constants in the `ItemQuoteBinConst` interface.

ItemQuoteBinConst.java

```
0  public interface ItemQuoteBinConst {
1    public static final String DEFAULT_ENCODING = "ISO_8859_1";
2    public static final int DISCOUNT_FLAG = 1 << 7;
3    public static final int IN_STOCK_FLAG =  1 << 0;
4    public static final int MAX_DESC_LEN = 255;
5    public static final int MAX_WIRE_LENGTH = 1024;
6  }
```

ItemQuoteBinConst.java

`ItemQuoteEncoderBin` implements the binary encoding.

ItemQuoteEncoderBin.java

```
0  import java.io.*;  // for ByteArrayOutputStream and DataOutputStream
1
2  public class ItemQuoteEncoderBin implements ItemQuoteEncoder, ItemQuoteBinConst {
3
4    private String encoding;  // Character encoding
5
6    public ItemQuoteEncoderBin() {
7      encoding = DEFAULT_ENCODING;
8    }
9
```

```
10    public ItemQuoteEncoderBin(String encoding) {
11      this.encoding = encoding;
12    }
13
14    public byte[] encode(ItemQuote item) throws Exception {
15
16      ByteArrayOutputStream buf = new ByteArrayOutputStream();
17      DataOutputStream out = new DataOutputStream(buf);
18      out.writeLong(item.itemNumber);
19      out.writeInt(item.quantity);
20      out.writeInt(item.unitPrice);
21      byte flags = 0;
22      if (item.discounted)
23        flags |= DISCOUNT_FLAG;
24      if (item.inStock)
25        flags |= IN_STOCK_FLAG;
26      out.writeByte(flags);
27      byte[] encodedDesc = item.itemDescription.getBytes(encoding);
28      if (encodedDesc.length > MAX_DESC_LEN)
29        throw new IOException("Item Description exceeds encoded length limit");
30      out.writeByte(encodedDesc.length);
31      out.write(encodedDesc);
32      out.flush();
33      return buf.toByteArray();
34    }
35  }
```

ItemQuoteEncoderBin.java

1. **Constants, variables, and constructors:** lines 4–12
2. `encode()`: lines 14–34
   - **Set up Output:** lines 16–17
     Again, a `ByteArrayOutputStream` collects the bytes of the encoded message. Encapsulating the `ByteArrayOutputStream` in a `DataOutputStream` allows use of its methods for writing binary integers.
   - **Write integers:** lines 18–20
     The `writeLong()` method writes the **long**'s 8 bytes to the stream in big-endian order. Similarly, `writeInt()` outputs 4 bytes.
   - **Write booleans as flags:** lines 21–26
     Encode each boolean using a single bit in a flag byte. Initialize the flag byte to 0, then set the appropriate bits to 1, if either *discounted* or *inStock* is true. (The bits are defined in the `ItemQuoteBinConst` interface to be the most and least significant bits of the byte, respectively.) Write the byte to the stream.
   - **Convert description string to bytes:** line 27
     Although `DataOutputStream` provides methods for writing `String`s, it supports only one

fixed encoding, namely, UTF-8. Because we want to support alternative encodings, we convert the string to bytes explicitly.

- **Check description length:** lines 28–29
  We are going to use an explicit length encoding for the string, with a single byte giving the length. The biggest value that byte can contain is 255 bytes, so the length of the encoded string must not exceed 255 bytes. If it does, we throw an exception.
- **Write encoded string:** lines 30–31
  Write the length of the encoded string, followed by the bytes in the buffer.
- **Flush output stream, return bytes:** line 32
  Ensure that all bytes are flushed from the `DataOutputStream` to the underlying byte buffer.

`ItemQuoteDecoderBin` implements the corresponding decoder function.

ItemQuoteDecoderBin.java

```
 0  import java.io.*;  // for ByteArrayInputStream
 1  import java.net.*; // for DatagramPacket
 2
 3  public class ItemQuoteDecoderBin implements ItemQuoteDecoder, ItemQuoteBinConst {
 4
 5    private String encoding;  // Character encoding
 6
 7    public ItemQuoteDecoderBin() {
 8      encoding = DEFAULT_ENCODING;
 9    }
10
11    public ItemQuoteDecoderBin(String encoding) {
12      this.encoding = encoding;
13    }
14
15    public ItemQuote decode(InputStream wire) throws IOException {
16      boolean discounted, inStock;
17      DataInputStream src = new DataInputStream(wire);
18      long itemNumber = src.readLong();
19      int quantity = src.readInt();
20      int unitPrice = src.readInt();
21      byte flags = src.readByte();
22      int stringLength = src.read(); // Returns an unsigned byte as an int
23      if (stringLength == -1)
24        throw new EOFException();
25      byte[] stringBuf = new byte[stringLength];
26      src.readFully(stringBuf);
27      String itemDesc = new String(stringBuf,encoding);
28      return new ItemQuote(itemNumber,itemDesc, quantity, unitPrice,
29        ((flags & DISCOUNT_FLAG) == DISCOUNT_FLAG),
30        ((flags & IN_STOCK_FLAG) == IN_STOCK_FLAG));
31    }
```

```
32
33    public ItemQuote decode(DatagramPacket p) throws IOException {
34      ByteArrayInputStream payload =
35        new ByteArrayInputStream(p.getData(), p.getOffset(), p.getLength());
36      return decode(payload);
37    }
38  }
```

ItemQuoteDecoderBin.java

1. **Constants, variables, and constructors:** lines 5–13
2. **Stream decode:** lines 15–31
   - **Wrap the** `InputStream`**:** line 17
     Using the given `InputStream`, construct a `DataInputStream` so we can make use of the methods `readLong()` and `readInt()` for reading binary data types from the input.
   - **Read integers:** lines 18–20
     Read the integers back in the same order they were written out. The `readLong()` method reads 8 bytes and constructs a (signed) **long** using big-endian byte ordering. The `readInt()` method reads 4 bytes and does the same thing. Either will throw an `EOFException` if the stream ends before the requisite number of bytes is read.
   - **Read flag byte:** line 21
     The flag byte is next; the values of the individual bits will be checked later.
   - **Read string length:** lines 22–24
     The next byte contains the length of the encoded string. Note that we use the `read()` method, which returns the contents of the next byte read as an integer between 0 and 255 (or −1), and that we read it into an **int**. If we read it into a **byte** (which is signed), we would not be able to distinguish between the case where the length is 255 and the case where the stream ends prematurely—both would return −1, since the signed interpretation of the 8-bit binary representation of 255 is −1.
   - **Allocate buffer and read encoded string:** lines 25–26
     Once we know how long the encoded string is, we allocate a buffer and call `readFully()`, which does not return until enough bytes have been read to fill the given buffer. `readFully()` will throw an `EOFException` if the stream ends before the buffer is filled. Note the advantage of the length-prefixed `String` representation: bytes do not have to be interpreted as characters until you have them all.
   - **Check flags:** lines 29–30
     The expressions used as parameters in the call to the constructor illustrate the standard method of checking whether a particular bit is set (equal to 1) in an integer type.
3. **Packet decode:** lines 33–37
   Simply wrap the packet's data in a `ByteArrayInputStream` and pass to the stream-decoding method.

3.4.3 Sending and Receiving

The encodings presented above can be used with both `Sockets` and `DatagramSockets`. We show the TCP usage first.

SendTCP.java

```
 0  import java.io.*;   // for Input/OutputStream
 1  import java.net.*;  // for Socket
 2
 3  public class SendTCP {
 4
 5    public static void main(String args[]) throws Exception {
 6
 7      if (args.length != 2)  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): <Destination> <Port>");
 9
10      InetAddress destAddr = InetAddress.getByName(args[0]);  // Destination address
11      int destPort = Integer.parseInt(args[1]);               // Destination port
12
13      Socket sock = new Socket(destAddr, destPort);
14
15      ItemQuote quote = new ItemQuote(1234567890987654L, "5mm Super Widgets",
16                                      1000, 12999, true, false);
17
18      // Send text-encoded quote
19      ItemQuoteEncoder coder = new ItemQuoteEncoderText();
20      byte[] codedQuote = coder.encode(quote);
21      System.out.println("Sending Text-Encoded Quote (" +
22                         codedQuote.length + " bytes): ");
23      System.out.println(quote);
24      sock.getOutputStream().write(codedQuote);
25
26      // Receive binary-encoded quote
27      ItemQuoteDecoder decoder = new ItemQuoteDecoderBin();
28      ItemQuote receivedQuote = decoder.decode(sock.getInputStream());
29      System.out.println("Received Binary-Encode Quote:");
30      System.out.println(receivedQuote);
31
32      sock.close();
33    }
34  }
```

1. `Socket` **setup:** line 13
2. **Send using text encoding:** lines 18–24
3. **Receive using binary encoding:** lines 26–30

RecvTCP.java

```
 0  import java.io.*;   // for Input/OutputStream
 1  import java.net.*;  // for Socket and ServerSocket
 2
 3  public class RecvTCP {
 4
 5    public static void main(String args[]) throws Exception {
 6
 7      if (args.length != 1)  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): <Port>");
 9
10      int port = Integer.parseInt(args[0]);   // Receiving Port
11
12      ServerSocket servSock = new ServerSocket(port);
13      Socket clntSock = servSock.accept();
14
15      // Receive text-encoded quote
16      ItemQuoteDecoder decoder = new ItemQuoteDecoderText();
17      ItemQuote quote = decoder.decode(clntSock.getInputStream());
18      System.out.println("Received Text-Encoded Quote:");
19      System.out.println(quote);
20
21      // Repeat quote with binary encoding, adding 10 cents to the price
22      ItemQuoteEncoder encoder = new ItemQuoteEncoderBin();
23      quote.unitPrice += 10;  // Add 10 cents to unit price
24      System.out.println("Sending (binary)...");
25      clntSock.getOutputStream().write(encoder.encode(quote));
26
27      clntSock.close();
28      servSock.close();
29    }
30  }
```

RecvTCP.java

1. `Socket` **setup:** line 12
2. **Accept client connection:** line 13

3. **Receive and print out a text-encoded message:** lines 15–19
4. **Send a binary-encoded message:** lines 21–25
   Note that before sending, we add 10 cents to the unit price given in the original message.

To demonstrate the use of the encoding and decoding classes with datagrams, we include a simple UDP sender and receiver. Since this is very similar to the TCP code, we do not include any code description.

SendUDP.java

```
 0  import java.net.*;  // for DatagramSocket, DatagramPacket, and InetAddress
 1  import java.io.*;   // for IOException
 2
 3  public class SendUDP {
 4
 5    public static void main(String args[]) throws Exception {
 6
 7      if (args.length != 2 && args.length != 3)  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): <Destination>" +
 9                   " <Port> [<encoding]");
10
11      InetAddress destAddr = InetAddress.getByName(args[0]);  // Destination address
12      int destPort = Integer.parseInt(args[1]);               // Destination port
13
14      ItemQuote quote = new ItemQuote(1234567890987654L, "5mm Super Widgets",
15                                      1000, 12999, true, false);
16
17      DatagramSocket sock = new DatagramSocket(); // UDP socket for sending
18
19      ItemQuoteEncoder encoder = (args.length == 3 ?
20          new ItemQuoteEncoderText(args[2]) :
21          new ItemQuoteEncoderText());
22
23      byte[] codedQuote = encoder.encode(quote);
24
25      DatagramPacket message = new DatagramPacket(codedQuote, codedQuote.length,
26                                                  destAddr, destPort);
27      sock.send(message);
28
29      sock.close();
30    }
31  }
```

SendUDP.java

RecvUDP.java

```
 0  import java.net.*;  // for DatagramSocket and DatagramPacket
 1  import java.io.*;   // for IOException
 2
 3  public class RecvUDP implements ItemQuoteTextConst {
 4
 5    public static void main(String[] args) throws Exception {
 6
 7      if (args.length != 1 && args.length != 2)  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): <Port> [<encoding>]");
 9
10      int port = Integer.parseInt(args[0]);   // Receiving Port
11
12      DatagramSocket sock = new DatagramSocket(port);  // UDP socket for receiving
13      ItemQuoteDecoder decoder = (args.length == 2 ?   // Which encoding
14                                  new ItemQuoteDecoderText(args[1]) :
15                                  new ItemQuoteDecoderText() );
16
17      DatagramPacket packet = new DatagramPacket(
18        new byte[MAX_WIRE_LENGTH], MAX_WIRE_LENGTH);
19      sock.receive(packet);
20
21      ItemQuote quote = decoder.decode(packet);
22      System.out.println(quote);
23
24      sock.close();
25    }
26  }
```

RecvUDP.java

3.5 Wrapping Up

We have seen how Java data types can be encoded in different ways, and how messages can be constructed from various types of information. You may be aware that recent versions of Java include *serialization* capabilities—the `Serializable` and `Externalizable` interfaces—which provide for instances of supporting Java classes to be converted to and from byte sequences very easily. It might seem that having these interfaces available would eliminate the need for what has been described above, and that is to some extent true. However, it is not always the case, for a couple of reasons.

First, the encoded forms produced by `Serializable` may not be very efficient. They may include information that is meaningless outside the context of the Java Virtual Machine (JVM), and may also incur overhead to provide flexibility that may not be needed. Second, `Serializable` and `Externalizable` cannot be used when a different wire format has already been specified—

for example, by a standardized protocol. And finally, custom-designed classes have to provide their own implementations of the serialization interfaces anyway.

A basic tenet of good protocol design is that the protocol should constrain the implementor as little as possible and should minimize assumptions about the platform on which the protocol will be implemented. We therefore avoid the use of `Serializable` and `Externalizable` in this book, and instead use more direct encoding and decoding methods.

3.6 Exercises

1. What happens if the `Encoder` uses a different encoding than the `Decoder`?
2. Positive integers larger than $2^{31} - 1$ cannot be represented as **int**s in Java, yet they can be represented as 32-bit binary numbers. Write a method to write such an integer to a stream. It should take a **long** and an **OutputStream** as parameters.
3. Rewrite the binary encoder so that the Item Description is terminated by "`\r\n`" instead of being length encoded. Use `Send/RecvTCP` to test this new encoding.
4. The `nextToken()` method of `DelimitedInputStream` assumes that either the delimiter or an end-of-stream (EoS) terminates a token; however, finding the EoS may be an error in some protocols. Rewrite `nextToken()` to include a second, boolean parameter. If the parameter value is `true`, then the EoS terminates a token without error; otherwise, the EoS generates an error.

This Page Intentionally Left Blank

chapter 4

Beyond the Basics

The client and server examples in Chapter 2 demonstrate the basic model for programming with sockets in Java. The next step is to apply these concepts in various programming models, such as multitasking, nonblocking I/O, and broadcasting.

4.1 Multitasking

Our basic TCP echo server from Chapter 2 handles one client at a time. If a client connects while another is already being serviced, the server will not echo the new client's data until it has finished with the current client, although the new client will be able to send data as soon as it connects. This type of server is known as an *iterative server*. Iterative servers handle clients sequentially, finishing with one client before servicing the next. They work best for applications where each client requires a small, bounded amount of server connection time; however, if the time to handle a client can be long, the wait experienced by subsequent clients may be unacceptable.

To demonstrate the problem, add a 10-second sleep using `Thread.sleep()` after the `Socket` constructor call in `TCPEchoClient.java` and experiment with several clients simultaneously accessing the TCP echo server. Here the sleep operation simulates an operation that takes significant time, such as slow file or network I/O. Note that a new client must wait for all already-connected clients to complete before it gets service.

What we need is some way for each connection to proceed independently, without interfering with other connections. Java threads provide exactly that: a convenient mechanism allowing servers to handle many clients simultaneously. Using threads, a single application can work on several tasks concurrently. In our echo server, we can give responsibility for each client to an independently executing thread. All of the examples we have seen so far consist of a single thread, which simply executes the `main()` method.

In this section we describe two approaches to coding *concurrent servers*, namely, *thread-per-client*, where a new thread is spawned to handle each client connection, and *thread pool*, where a fixed, prespawned set of threads work together to handle client connections.

4.1.1 Java Threads

Java provides two approaches for performing a task in a new thread: 1) defining a subclass of the `Thread` class with a `run()` method that performs the task, and instantiating it; or 2) defining a class that implements the `Runnable` interface with a `run()` method that performs the task, and passing an instance of that class to the `Thread` constructor. In either case, the new thread does not begin execution until its `start()` method is invoked. The first approach can only be used for classes that do not already extend some other class; therefore, we focus on the second approach, which is always applicable. The `Runnable` interface contains a single method prototype:

```
public void run();
```

When the `start()` method of an instance of `Thread` is invoked, the JVM causes the instance's `run()` method to be executed in a new thread, concurrently with all others. Meanwhile, the *original* thread returns from its call to `start()` and continues its execution independently. (Note that directly calling the `run()` method of a `Thread` or `Runnable` instance has the normal procedure-call semantics: the method is executed in the caller's thread.) The exact interleaving of thread execution is determined by several factors, including the implementation of the JVM, the load, the underlying OS, and the host configuration. For example, on a uniprocessor system, threads share the processor sequentially; on a multiprocessor system, multiple threads from the same application can run simultaneously on different processors.

In the following example, `ThreadExample.java` implements the `Runnable` interface with a `run()` method that repeatedly prints a greeting to the system output stream.

ThreadExample.java

```
 0  public class ThreadExample implements Runnable {
 1
 2    private String greeting;   // Message to print to console
 3
 4    public ThreadExample(String greeting) {
 5      this.greeting = greeting;
 6    }
 7
 8    public void run() {
 9      for (;;) {
10        System.out.println(Thread.currentThread().getName() + ":  " + greeting);
11        try {
12          Thread.sleep((long) (Math.random() * 100)); // Sleep 0 to 100 milliseconds
13        } catch (InterruptedException e) {}  // Will not happen
14      }
15    }
16
17    public static void main(String[] args) {
18      new Thread(new ThreadExample("Hello")).start();
```

```
19      new Thread(new ThreadExample("Aloha")).start();
20      new Thread(new ThreadExample("Ciao")).start();
21    }
22  }
```

ThreadExample.java

1. **Declaration of implementation of the `Runnable` interface:** line 0
   Since `ThreadExample` implements the `Runnable` interface, it can be passed to the constructor of `Thread`. If `ThreadExample` fails to provide a `run()` method, the compiler will complain.
2. **Member variables and constructor:** lines 2–6
   Each instance of `ThreadExample` contains its own greeting string.
3. `run()`**:** lines 8–15
   Loop forever performing:
   - **Print the thread name and instance greeting:** line 10
     The static method `Thread.currentThread()` returns a reference to the thread from which it is called, and `getName()` returns a string containing the name of that thread.
   - **Suspend thread:** lines 11–13
     After printing its instance's greeting message, each thread sleeps for a random amount of time (between 0 and 100 milliseconds) by calling the static method `Thread.sleep()`, which takes the number of milliseconds to sleep as a parameter. `Math.random()` returns a random **double** between 0.0 and 1.0. `Thread.sleep()` can be interrupted by another thread, in which case an `InterruptedException` is thrown. Our example does not include an interrupt call, so the exception will not happen in this application.
4. `main()`**:** lines 17–21
   Each of the three statements in `main()` does the following: 1) creates a new instance of `ThreadExample` with a different greeting string, 2) passes this new instance to the constructor of `Thread`, and 3) calls the new `Thread` instance's `start()` method. Each thread independently executes the `run()` method of `ThreadExample`, while the `main()` thread terminates. Note that the JVM does not terminate until all nondaemon (see `Threads` API) threads terminate.

Upon execution, an interleaving of the three greeting messages is printed to the console. The exact interleaving of the numbers depends upon the factors mentioned earlier.

4.1.2 Server Protocol

Since the two server approaches we are going to describe (thread-per-client and thread pool) are independent of the particular client-server protocol, we want to be able to use the same protocol code for both. The code for the echo protocol is given in the class `EchoProtocol`, which encapsulates the implementation of the server side of the echo protocol. The idea is that the server creates a separate instance of `EchoProtocol` for each connection, and protocol

execution begins when `run()` is called on an instance. The code in `run()` is almost identical to the connection handling code in `TCPEchoServer.java`, except that a logging capability (described shortly) has been added. The class implements the `Runnable` interface, so we can create a thread that independently executes `run()`, or we can invoke `run()` directly.

EchoProtocol.java

```
 0  import java.net.*;   // for Socket
 1  import java.io.*;    // for IOException and Input/OutputStream
 2  import java.util.*;  // for ArrayList
 3
 4  class EchoProtocol implements Runnable {
 5    static public final int BUFSIZE = 32;   // Size (in bytes) of I/O buffer
 6
 7    private Socket clntSock;  // Connection socket
 8    private Logger logger;    // Logging facility
 9
10    public EchoProtocol(Socket clntSock, Logger logger) {
11      this.clntSock = clntSock;
12      this.logger = logger;
13    }
14
15    public void run() {
16      ArrayList entry = new ArrayList();
17      entry.add("Client address and port = " +
18        clntSock.getInetAddress().getHostAddress() + ":" +
19        clntSock.getPort());
20      entry.add("Thread = " + Thread.currentThread().getName());
21
22      try {
23        // Get the input and output I/O streams from socket
24        InputStream in = clntSock.getInputStream();
25        OutputStream out = clntSock.getOutputStream();
26
27        int recvMsgSize;                        // Size of received message
28        int totalBytesEchoed = 0;               // Bytes received from client
29        byte[] echoBuffer = new byte[BUFSIZE];  // Receive Buffer
30        // Receive until client closes connection, indicated by -1
31        while ((recvMsgSize = in.read(echoBuffer)) != -1) {
32          out.write(echoBuffer, 0, recvMsgSize);
33          totalBytesEchoed += recvMsgSize;
34        }
35
36        entry.add("Client finished; echoed " + totalBytesEchoed + " bytes.");
37      } catch (IOException e) {
38        entry.add("Exception = " +  e.getMessage());
39      }
```

```
40
41      try {  // Close socket
42        clntSock.close();
43      } catch (IOException e) {
44        entry.add("Exception = " +  e.getMessage());
45      }
46
47      logger.writeEntry(entry);
48    }
49  }
```

EchoProtocol.java

1. **Declaration of implementation of the** `Runnable` **interface:** line 4
2. **Member variables and constructor:** lines 7–13
   Each instance of `EchoProtocol` contains a socket for the connection and a reference to the logger.
3. `run()`**:** lines 15–48
   Implement the echo protocol:
   - **Write the client and thread information to a buffer:** lines 16–20
     `ArrayList` is a dynamically sized container of `Objects`. The `add()` method of `ArrayList` inserts the specified object at the end of the list. In this case, the inserted object is a `String`. Each element of the `ArrayList` represents a line of output to the logger.
   - **Execute the echo protocol:** lines 22–45
   - **Write the elements (one per line) of the** `ArrayList` **instance to the logger:** line 47

The logger allows for synchronized reporting of thread creation and client completion, so that entries from different threads are not interleaved. This facility is defined by the `Logger` interface, which has methods for writing strings or object collections.

Logger.java

```
0  import java.util.*;  // for Collection
1
2  public interface Logger {
3    public void writeEntry(Collection entry); // Write list of lines
4    public void writeEntry(String entry);     // Write single line
5  }
```

Logger.java

`writeEntry()` logs the given string or object collection. How it is logged depends on the implementation. One possibility is to send the log messages to the console.

ConsoleLogger.java

```
 0  import java.util.*;  // for Collection and Iterator
 1
 2  class ConsoleLogger implements Logger {
 3    public synchronized void writeEntry(Collection entry) {
 4      for (Iterator line = entry.iterator(); line.hasNext();)
 5        System.out.println(line.next());
 6      System.out.println();
 7    }
 8
 9    public synchronized void writeEntry(String entry) {
10      System.out.println(entry);
11      System.out.println();
12    }
13  }
```

ConsoleLogger.java

Another possibility is to write the log messages to a file specified in the constructor, as in the following:

FileLogger.java

```
 0  import java.io.*;    // for PrintWriter and FileWriter
 1  import java.util.*;  // for Collection and Iterator
 2
 3  class FileLogger implements Logger {
 4
 5    PrintWriter out;  // Log file
 6
 7    public FileLogger(String filename) throws IOException {
 8      out = new PrintWriter(new FileWriter(filename), true);  // Create log file
 9    }
10
11    public synchronized void writeEntry(Collection entry) {
12      for (Iterator line = entry.iterator(); line.hasNext();)
13        out.println(line.next());
14      out.println();
15    }
16
```

```
17    public synchronized void writeEntry(String entry) {
18      out.println(entry);
19      out.println();
20    }
21  }
```

FileLogger.java

We are now ready to introduce some different approaches to concurrent servers.

4.1.3 Thread-per-Client

In a *thread-per-client* server, a new thread is created to handle each connection. The server executes a loop that runs forever, listening for connections on a specified port and repeatedly accepting an incoming connection from a client and then spawning a new thread to handle that connection.

`TCPEchoServerThread.java` implements the thread-per-client architecture. It is very similar to the iterative server, using a single indefinite loop to receive and process client requests. The main difference is that it creates a thread to handle the connection instead of handling it directly. (This is possible because `EchoProtocol` implements the `Runnable` interface.)

TCPEchoServerThread.java

```
 0  import java.net.*;  // for Socket, ServerSocket, and InetAddress
 1  import java.io.*;   // for IOException and Input/OutputStream
 2
 3  public class TCPEchoServerThread {
 4
 5    public static void main(String[] args) throws IOException {
 6
 7      if (args.length != 1)  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): <Port>");
 9
10      int echoServPort = Integer.parseInt(args[0]);  // Server port
11
12      // Create a server socket to accept client connection requests
13      ServerSocket servSock = new ServerSocket(echoServPort);
14
15      Logger logger = new ConsoleLogger();       // Log messages to console
16
17      // Run forever, accepting and spawning threads to service each connection
18      for (;;) {
19        try {
20          Socket clntSock = servSock.accept();  // Block waiting for connection
```

```
21          EchoProtocol protocol = new EchoProtocol(clntSock, logger);
22          Thread thread = new Thread(protocol);
23          thread.start();
24          logger.writeEntry("Created and started Thread = " + thread.getName());
25        } catch (IOException e) {
26          logger.writeEntry("Exception = " + e.getMessage());
27        }
28      }
29      /* NOT REACHED */
30    }
31  }
```

TCPEchoServerThread.java

1. **Parameter parsing and server socket/logger creation:** lines 7–15
2. **Loop forever, handling incoming connections:** lines 17–28
   - **Accept an incoming connection:** line 20
   - **Create a protocol instance to handle new connection:** line 21
     Each connection gets its own instance of `EchoProtocol`. Each instance maintains the state of its particular connection. The echo protocol has little internal state, but more sophisticated protocols may require substantial amounts of state.
   - **Create, start, and log a new thread for the connection:** lines 22–24
     Since `EchoProtocol` implements the `Runnable` interface, we can give our new instance to the `Thread` constructor, and the new thread will execute the `run()` method of `EchoProtocol` when `start()` is invoked. The `getName()` method of `Thread` returns a `String` containing a name for the new thread.
   - **Handle exception from `accept()`:** lines 25–27
     If some I/O error occurs, `accept()` throws an `IOException`. In our earlier iterative echo server (`TCPEchoServer.java`), the exception is not handled, and such an error terminates the server. Here we handle the exception by logging the error and continuing execution.

4.1.4 Factoring the Server

Our threaded server does what we want it to, but the code is not very reusable or extensible. First, the echo protocol is hard-coded in the server. What if we want an HTTP server instead? We could write an `HTTPProtocol` and replace the instantiation of `EchoProtocol` in `main()`; however, we would have to revise `main()` and have a separate main class for each different protocol that we implement.

We want to be able to instantiate a protocol instance of the appropriate type for each connection without knowing any specifics about the protocol, including the name of a constructor. This problem—instantiating an object without knowing details about its type—arises frequently in object-oriented programming, and there is a standard solution: use a *factory*. A

factory object supplies instances of a particular class, hiding the details of how the instance is created, such as what constructor is used.

For our protocol factory, we define the `ProtocolFactory` interface to have a single method, `createProtocol()`, which takes `Socket` and `Logger` instances as arguments and returns an instance implementing the desired protocol. Our protocols will all implement the `Runnable` interface, so that once we have an instance we can simply call `run()` (or `start()` on a `Thread` constructed from the protocol instance) to execute the protocol for that connection. Thus, our protocol factory returns instances that implement the `Runnable` interface:

ProtocolFactory.java

```
0  import java.net.*;  // for Socket
1
2  public interface ProtocolFactory {
3    public Runnable createProtocol(Socket clntSock, Logger logger);
4  }
```

ProtocolFactory.java

We now need to implement a protocol factory for the echo protocol. The factory class is simple. All it does is return a new instance of `EchoProtocol` whenever `createProtocol()` is called.

EchoProtocolFactory.java

```
0  import java.net.*;  // for Socket
1
2  public class EchoProtocolFactory implements ProtocolFactory {
3    public Runnable createProtocol(Socket clntSock, Logger logger) {
4      return new EchoProtocol(clntSock, logger);
5    }
6  }
```

EchoProtocolFactory.java

We have factored out some of the details of protocol instance creation from our server, so that the various iterative and concurrent servers can reuse the protocol code. However, the server approach (iterative, thread-per-client, etc.) is still hard-coded in the `main()`. These server approaches deal with how to *dispatch* each connection to the appropriate handling mechanism. To provide greater extensibility, we want to factor out the dispatching model from the `main()` of `TCPEchoServerThread.java` so that we can use any dispatching model with any protocol. Since we have many potential dispatching models, we define the `Dispatcher` interface to hide the particulars of the threading strategy from the rest of the server code. It contains a

single method, `startDispatching()`, which tells the dispatcher to start handling clients accepted via the given `ServerSocket`, creating protocol instances using the given `ProtocolFactory`, and logging via the given `Logger`.

Dispatcher.java

```
 0  import java.net.*;  // for ServerSocket
 1
 2  public interface Dispatcher {
 3    public void startDispatching(ServerSocket servSock, Logger logger,
 4                                 ProtocolFactory protoFactory);
 5  }
```

Dispatcher.java

To implement the thread-per-client dispatcher, we simply pull the for loop from `main()` in `TCPEchoServerThread.java` into the `startDispatching()` method of the new dispatcher. The only other change we need to make is to use the protocol factory instead of instantiating a particular protocol.

ThreadPerDispatcher.java

```
 0  import java.net.*;  // for Socket and ServerSocket
 1  import java.io.*;   // for IOException
 2
 3  class ThreadPerDispatcher implements Dispatcher {
 4
 5    public void startDispatching(ServerSocket servSock, Logger logger,
 6                                 ProtocolFactory protoFactory) {
 7      // Run forever, accepting and spawning threads to service each connection
 8      for (;;) {
 9        try {
10          Socket clntSock = servSock.accept();  // Block waiting for connection
11          Runnable protocol = protoFactory.createProtocol(clntSock, logger);
12          Thread thread = new Thread(protocol);
13          thread.start();
14          logger.writeEntry("Created and started Thread = " + thread.getName());
15        } catch (IOException e) {
16          logger.writeEntry("Exception = " + e.getMessage());
17        }
18      }
19      /* NOT REACHED */
20    }
21  }
```

ThreadPerDispatcher.java

We demonstrate the use of this dispatcher and protocol factory in `ThreadMain.java`, which we introduce after discussing the thread-pool approach to dispatching.

4.1.5 Thread Pool

Every new thread consumes system resources: spawning a thread takes CPU cycles and each thread has its own data structures (e.g., stacks) that consume system memory. In addition, the scheduling and context switching among threads creates extra work. As the number of threads increases, more and more system resources are consumed by thread overhead. Eventually, the system is spending more time dealing with thread management than with servicing connections. At that point, adding an additional thread may actually *increase* client service time.

We can avoid this problem by limiting the total number of threads and reusing threads. Instead of spawning a new thread for each connection, the server creates a *thread pool* on start-up by spawning a fixed number of threads. When a new client connection arrives at the server, it is assigned to a thread from the pool. When the thread finishes with the client, it returns to the pool, ready to handle another request. Connection requests that arrive when all threads in the pool are busy are queued to be serviced by the next available thread.

Like the thread-per-client server, a thread-pool server begins by creating a `ServerSocket`. Then it spawns N threads, each of which loops forever, accepting connections from the (shared) `ServerSocket` instance. When multiple threads simultaneously call `accept()` on the same `ServerSocket` instance, they all block until a connection is established. Then the system selects one thread, and the `Socket` instance for the new connection is returned *only in that thread*. The other threads remain blocked until the next connection is established and another lucky winner is chosen.

Since each thread in the pool loops forever, processing connections one by one, a thread-pool server is really a set of iterative servers. Unlike the thread-per-client server, a thread-pool thread does not terminate when it finishes with a client. Instead, it starts over again, blocking on `accept()`.

A thread pool is simply a different model for dispatching connection requests, so all we really need to do is write another dispatcher. `PoolDispatcher.java` implements our thread-pool dispatcher. To see how the thread-pool server would be implemented without dispatchers and protocol factories, see `TCPEchoServerPool.java` on the book's Web site.

PoolDispatcher.java

```
0  import java.net.*;  // for Socket and ServerSocket
1  import java.io.*;   // for IOException
2
3  class PoolDispatcher implements Dispatcher {
4
5    static final String NUMTHREADS = "8";        // Default thread-pool size
6    static final String THREADPROP = "Threads";  // Name of thread property
7
8    private int numThreads;           // Number of threads in pool
```

```
 9
10    public PoolDispatcher() {
11      // Get the number of threads from the System properties or take the default
12      numThreads = Integer.parseInt(System.getProperty(THREADPROP, NUMTHREADS));
13    }
14
15    public void startDispatching(final ServerSocket servSock, final Logger logger,
16                                 final ProtocolFactory protoFactory) {
17      // Create N-1 threads, each running an iterative server
18      for (int i = 0; i < (numThreads - 1); i++) {
19        Thread thread = new Thread() {
20          public void run() {
21            dispatchLoop(servSock, logger, protoFactory);
22          }
23        };
24        thread.start();
25        logger.writeEntry("Created and started Thread = " + thread.getName());
26      }
27      logger.writeEntry("Iterative server starting in main thread " +
28                        Thread.currentThread().getName());
29      // Use main thread as Nth iterative server
30      dispatchLoop(servSock, logger, protoFactory);
31      /* NOT REACHED */
32    }
33
34    private void dispatchLoop(ServerSocket servSock, Logger logger,
35                              ProtocolFactory protoFactory) {
36      // Run forever, accepting and handling each connection
37      for (;;) {
38        try {
39          Socket clntSock = servSock.accept();  // Block waiting for connection
40          Runnable protocol = protoFactory.createProtocol(clntSock, logger);
41          protocol.run();
42        } catch (IOException e) {
43          logger.writeEntry("Exception = " +  e.getMessage());
44        }
45      }
46    }
47  }
```

PoolDispatcher.java

1. `PoolDispatcher()`: lines 10–13
   The thread-pool solution needs an additional piece of information: the number of threads in the pool. We need to provide this information to the instance before the thread pool is constructed. We could pass the number of threads to the constructor, but this limits our options because the constructor interface varies by dispatcher. We use system properties

to specify the number of threads to PoolDispatcher. The call to System.getProperty() returns a String containing the value of the "Threads" property or the default value if the property is not defined. The string is then converted to an integer. (See the discussion of system properties in the text below.)

2. startDispatching(): lines 15–32
   - **Spawn N – 1 threads to execute** dispatchLoop(): lines 17–26
     For each loop iteration, an instance of an anonymous class that extends Thread is created. When the start() method is called, the thread executes the run() method of this anonymous class. The run() method simply calls dispatchLoop(), which implements an iterative server.
   - **Execute** dispatchLoop() **in the main thread:** lines 27–30
     The original calling thread serves as the *N*th thread of the pool.
3. dispatchLoop(): lines 34–46
   - **Accept an incoming connection:** line 39
     Since there are *N* threads executing dispatchLoop(), up to *N* threads can be blocked on *servSock*'s accept(), waiting for an incoming connection. The system ensures that only one thread gets a Socket for any particular connection. If no threads are blocked on accept() when a client connection is established, the new connection is queued until the next call to accept() (see Section 5.4.1).
   - **Create a protocol instance to handle new connection:** line 40
   - **Run the protocol for the connection:** line 41
   - **Handle exception from** accept(): lines 42–44

Since threads are reused, the thread-pool solution only pays the overhead of thread creation *N* times, irrespective of the total number of client connections. Since we control the maximum number of simultaneously executing threads, we can control scheduling overhead. Of course, if we spawn too few threads, we can still have clients waiting a long time for service; therefore, the size of the thread pool should be tuned so that client connection time is minimized.

In PoolDispatcher.java, we used system properties to specify the number of threads in the pool. Here, we give a brief description of how system properties work.. The System class contains a Properties instance that holds a set of externally defined property/value pairs (e.g., class path and JVM version). We can also define our own properties. For example, we might want to know a user's favorite color. We could place this information in the "user.favoritecolor" property. The following code demonstrates how to fetch and print out all system properties, using the getProperties() method of System, and how to find a particular property value, using System.getProperty(). The second parameter to getProperty() specifies a value ("None"), to be used if the property is not found. (See ListProperties.java on the book's Web site for a complete example.)

```
System.getProperties().list(System.out);   // Print all System properties
System.out.println("\nFavorite Color: " +  // Print favorite color property
      System.getProperty("user.favoritecolor", "None"));
```

When running Java programs from the command line, we simply use the –D option to set a property value. For example, to set the property "user.favoritecolor" to "blue," we would try

```
% java -Duser.favoritecolor=blue ListProperties
```

Note that properties are typically defined with hierarchical (general to specific) names, such as *java.class.path*. For brevity's sake, we did not use such a name, but in a production application hierarchical names should be used to avoid collisions.

The `main()` of `ThreadMain.java` demonstrates how to use either the thread-per-client or thread-pool server. This application takes three parameters: 1) the port number for the server, 2) the protocol name (use "Echo" for the echo protocol), and 3) the dispatcher name (use "ThreadPer" or "Pool" for the thread-per-client and thread-pool servers, respectively). The number of threads for the thread pool defaults to 8. However, this can be changed to 4, for example, by setting a system property using the `–DThreads=4` option to the JVM.

```
% java -DThreads=4 ThreadMain 5000 Echo Pool
```

Note that you must compile `EchoProtocolFactory.java`, `ThreadPerDispatcher.java`, and `PoolDispatcher.java` explicitly before running `ThreadMain.java`. Failure to do so will result in a `ClassNotFoundException`. Those classes are not referenced by name in `ThreadMain` (that's the idea!), so they will not be automatically compiled with `ThreadMain`.

ThreadMain.java

```
 0  import java.net.*;  // for ServerSocket
 1  import java.io.*;   // for IOException
 2
 3  public class ThreadMain {
 4
 5    public static void main(String[] args) throws Exception {
 6
 7      if (args.length != 3)  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): [<Optional properties>]"
 9                                           + " <Port> <Protocol> <Dispatcher>");
10
11      int servPort = Integer.parseInt(args[0]);  // Server Port
12      String protocolName = args[1];             // Protocol name
13      String dispatcherName = args[2];           // Dispatcher name
14
15      ServerSocket servSock = new ServerSocket(servPort);
16      Logger logger = new ConsoleLogger();       // Log messages to console
17      ProtocolFactory protoFactory = (ProtocolFactory)  // Get protocol factory
18        Class.forName(protocolName + "ProtocolFactory").newInstance();
19      Dispatcher dispatcher = (Dispatcher)       // Get dispatcher
20        Class.forName(dispatcherName + "Dispatcher").newInstance();
21
```

```
22      dispatcher.startDispatching(servSock, logger, protoFactory);
23      /* NOT REACHED */
24    }
25  }
```

ThreadMain.java

1. **Application setup and parameter parsing:** lines 0–13
2. **Create server socket and logger:** lines 15–16
3. **Instantiate a protocol factory:** lines 17–18
   The protocol name is passed as the second parameter. We adopt the naming convention of `<ProtocolName>ProtocolFactory` for the class name of the factory for the protocol name `<ProtocolName>`. For example, if the second parameter is "Echo," the corresponding protocol factory is `EchoProtocolFactory`. The static method `Class.forName()` takes the name of a class and returns a `Class` object. The `newInstance()` method of `Class` creates a new instance of the class using the parameterless constructor. *protoFactory* refers to this new instance of the specified protocol factory.
4. **Instantiate a dispatcher:** lines 19–20
   The dispatcher name is passed as the third parameter. We adopt the naming convention of `<DispatcherType>Dispatcher` for the class name of the dispatcher of type `<Dispatcher-Type>`. For example, if the third parameter is "ThreadPer," the corresponding dispatcher is `ThreadPerDispatcher`. *dispatcher* refers to the new instance of the specified dispatcher.
5. **Start dispatching clients:** line 22

`ThreadMain.java` makes it easy to use other protocols and dispatchers. The book's Web site contains some additional examples. For example, see `TimeProtocolFactory.java` for an implementation of the time protocol where clients can get the server time by simply connecting to the server on the time port.

See `GUIThreadMain.java` on the book's Web site for an example of server integration with a GUI. This application lists the currently connected client. You will need a protocol-specific GUI implementation (see `GUIEchoProtocolFactory.java`).[1] The parameters to this application are the same as for `ThreadMain`. For this application to work, you must specify the GUI version of the protocol factory on the command line (e.g., `GUIEcho` instead of `Echo`).

4.2 Nonblocking I/O

Socket I/O calls may block for several reasons. Data input methods `read()` and `receive()` block if data is not available. A `write()` on a TCP socket may block if there is not sufficient space to buffer the transmitted data. The `accept()` method of `ServerSocket` and the `Socket` constructor

[1] Clearly, more decomposition is possible, but it is beyond the scope of this book.

both block until a connection has been established (see Section 5.4). Meanwhile, long round-trip times, high error rate connections, and slow (or deceased) servers may cause connection establishment to take a long time. In all of these cases, the method returns only after the request has been satisfied. Of course, a blocking method call halts the execution of the application.

What about a program that has other tasks to perform while waiting for call completion (e.g., updating the "busy" cursor or responding to user requests)? These programs may have no time to wait on a blocked method call. Or what about lost UDP datagrams? In `UDPEchoClientTimeout.java`, the client sends a datagram to the server and then waits to receive a response. If a datagram is not received before the timer expires, `receive()` unblocks to allow the client to handle the datagram loss. Here we describe the general nonblocking approaches (where they exist) for various I/O methods. (*Note:* As this book goes to press, additional nonblocking I/O features have been proposed for version 1.4 of the JDK. Because these features are still under development, we do not cover them here.)

4.2.1 `accept()`, `read()`, and `receive()`

For these methods, we can set a bound on the maximum time (in milliseconds) to block, using the `setSoTimeout()` method of `Socket`, `ServerSocket`, and `DatagramSocket`. If the specified time elapses before the method returns, an `InterruptedIOException` is thrown. For `Socket` instances, we can also use the `available()` method of the socket's `InputStream` to check for available data before calling `read()`.

4.2.2 Connecting and Writing

The `Socket` constructor attempts to establish a connection to the host and port supplied as arguments, blocking until either the connection is established or a system-imposed timeout occurs. Unfortunately, the system-imposed timeout is long (on the order of minutes), and Java does not provide any means of shortening it.

A `write()` call blocks until the last byte written is copied into the TCP implementation's local buffer; if the available buffer space is smaller than the size of the write, some data must be successfully transferred to the other end of the connection before the call to `write()` will return (see Section 5.1 for details). Thus, the amount of time that a `write()` may block is controlled by the receiving application. Unfortunately, Java currently does not provide any way to cause a `write()` to time out, nor can it be interrupted by another thread. Therefore, any protocol that sends a large enough amount of data over a `Socket` instance can block for an unbounded amount of time. (See Section 5.2 for further discussion on the consequences.)

4.2.3 Limiting Per-Client Time

Suppose we want to implement the Echo protocol with a limit on the amount of time taken to service each client. That is, we define a target, TIMELIMIT, and implement the protocol in such a way that after TIMELIMIT milliseconds, the protocol instance is terminated. One approach

simply has the protocol instance keep track of the amount of the remaining time, and use `setSoTimeout()` to ensure that no `read()` call blocks for longer than that time. Unfortunately, there is no way to bound the duration of a `write()` call, so we cannot really guarantee that the time limit will hold. `TimelimitEchoProtocolFactory.java` implements this approach.

TimelimitEchoProtocolFactory.java

```
 0  import java.net.*;   // for Socket
 1  import java.io.*;    // for IOException and Input/OutputStream
 2  import java.util.*;  // for ArrayList
 3
 4  public class TimelimitEchoProtocolFactory implements ProtocolFactory {
 5
 6    public Runnable createProtocol(Socket clntSock, Logger logger) {
 7      return new TimelimitEchoProtocol(clntSock, logger);
 8    }
 9  }
10
11  class TimelimitEchoProtocol implements Runnable {
12    private static final int BUFSIZE = 32;  // Size (in bytes) of receive buffer
13    private static final String TIMELIMIT = "10000";  // Default time limit (ms)
14    private static final String TIMELIMITPROP = "Timelimit";  // Thread property
15
16    private int timelimit;
17    private Socket clntSock;
18    private Logger logger;
19
20    public TimelimitEchoProtocol(Socket clntSock, Logger logger) {
21      this.clntSock = clntSock;
22      this.logger = logger;
23      // Get the time limit from the System properties or take the default
24      timelimit = Integer.parseInt(System.getProperty(TIMELIMITPROP, TIMELIMIT));
25    }
26
27    public void run() {
28      ArrayList entry = new ArrayList();
29      entry.add("Client address and port = " +
30        clntSock.getInetAddress().getHostAddress() + ":" +
31        clntSock.getPort());
32      entry.add("Thread = " + Thread.currentThread().getName());
33
34      try {
35        // Get the input and output I/O streams from socket
36        InputStream in = clntSock.getInputStream();
37        OutputStream out = clntSock.getOutputStream();
38
```

```
39        int recvMsgSize;                        // Size of received message
40        int totalBytesEchoed = 0;               // Bytes received from client
41        byte[] echoBuffer = new byte[BUFSIZE];  // Receive buffer
42        long endTime = System.currentTimeMillis() + timelimit;
43        int timeBoundMillis = timelimit;
44
45        clntSock.setSoTimeout(timeBoundMillis);
46
47        // Receive until client closes connection, indicated by -1
48        while ((timeBoundMillis > 0) &&     // catch zero values
49               ((recvMsgSize = in.read(echoBuffer)) != -1)) {
50          out.write(echoBuffer, 0, recvMsgSize);
51          totalBytesEchoed += recvMsgSize;
52          timeBoundMillis = (int) (endTime - System.currentTimeMillis()) ;
53          clntSock.setSoTimeout(timeBoundMillis);
54        }
55
56        entry.add("Client finished; echoed " + totalBytesEchoed + " bytes.");
57      } catch (InterruptedIOException dummy) {
58        entry.add("Read timed out");
59      } catch (IOException e) {
60        entry.add("Exception = " +  e.getMessage());
61      }
62
63      try {  // Close socket
64        clntSock.close();
65      } catch (IOException e) {
66        entry.add("Exception = " +  e.getMessage());
67      }
68
69      logger.writeEntry(entry);
70    }
71  }
```

TimelimitEchoProtocolFactory.java

`TimelimitEchoProtocolFactory.java` contains both the factory and protocol instance classes. The factory is exactly like `EchoProtocolFactory`, with the exception that it instantiates `TimelimitEchoProtocol` instead of `EchoProtocol`. The `TimelimitEchoProtocol` class is similar to the `EchoProtocol` class, except that it attempts to bound the total time an echo connection can exist. The default time is 10 seconds; the total number of milliseconds per connection can be set using the "Timelimit" property.

Another approach to limiting client service time involves starting two threads per client: one that executes the protocol and another that acts as a "watchdog," sleeping until TIMELIMIT milliseconds pass or the other (protocol) thread finishes and interrupts it, whichever comes

first. If the watchdog awakens and the protocol thread has not finished, the watchdog terminates the protocol thread. Unfortunately, threads killing other threads is deprecated in Java, because the victim thread's abrupt termination might leave some objects in an inconsistent or unrecoverable state. Since there is no other way to interrupt a blocking `write()`, this solution usually will not work.

Finally, note that we could attempt to use nonblocking I/O instead of threads. Be warned, however, that these solutions typically involve polling loops employing busy-waiting. While adding threads does consume extra CPU and memory resources, the overhead is generally small, especially compared to that of busy-waiting.

4.3 Multiple Recipients

So far all of our sockets have dealt with communication between exactly two entities, usually a server and a client. Such one-to-one communication is sometimes called *unicast*. Some information is of interest to multiple recipients. In such cases, we could unicast a copy of the data to each recipient, but this may be very inefficient. Unicasting multiple copies over a single network connection wastes bandwidth by sending the same information multiple times. In fact, if we want to send data at a fixed rate, the bandwidth of our network connection defines a hard limit on the number of receivers we can support. For example, if our video server sends 1Mbps streams and its network connection is only 3Mbps (a healthy connection rate), we can only support three simultaneous users.

Fortunately, networks provide a way to use bandwidth more efficiently. Instead of making the sender responsible for duplicating packets, we can give this job to the network. In our video server example, we send a single copy of the stream across the server's connection to the network, which then duplicates the data only when appropriate. With this model of duplication, the server uses only 1Mbps across its connection to the network, irrespective of the number of clients.

There are two types of one-to-many service: *broadcast* and *multicast*. With broadcast, all hosts on the (local) network receive a copy of the message. With multicast, the message is sent to a *multicast address*, and the network delivers it only to those hosts that have indicated that they want to receive messages sent to that address. In general, only UDP sockets are allowed to broadcast or multicast.

4.3.1 Broadcast

Broadcasting UDP datagrams is similar to unicasting datagrams, except that a *broadcast address* is used instead of a regular (unicast) IP address. The *local broadcast* address (255.255.255.255) sends the message to every host on the same broadcast network. Local broadcast messages are never forwarded by routers. A host on an Ethernet network can send a message to all other hosts on that same Ethernet, but the message will not be forwarded by a router. IP also specifies *directed broadcast* addresses, which allow broadcasts to all hosts on a specified network; however, since most Internet routers do not forward directed broadcasts, we do not deal with them here.

There is no networkwide broadcast address that can be used to send a message to all hosts. To see why, consider the impact of a broadcast to every host on the Internet. Sending a single datagram would result in a very, very large number of packet duplications by the routers, and bandwidth would be consumed on each and every network. The consequences of misuse (malicious or accidental) are too great, so the designers of IP left such an Internetwide broadcast facility out on purpose.

Even so, local broadcast can be very useful. Often, it is used in state exchange for network games where the players are all on the same local (broadcast) network. In Java, the code for unicasting and broadcasting is the same. To play with broadcasting applications, simply run `SendUDP.java` using a broadcast destination address. Run `RecvUDP.java` as you did before (except that you can run several receivers at one time). *Caveat:* Some operating systems do not give regular users permission to broadcast, in which case this will not work.

4.3.2 Multicast

As with broadcast, the main difference between multicast and unicast is the form of the address. A multicast address identifies a set of receivers. The designers of IP allocated a range of the address space (from 224.0.0.0 to 239.255.255.255) dedicated to multicast. With the exception of a few reserved multicast addresses, a sender can send datagrams addressed to any address in this range. In Java, multicast applications generally communicate using an instance of `MulticastSocket`, a subclass of `DatagramSocket`. It is important to understand that a `MulticastSocket` is actually a UDP socket (`DatagramSocket`), with some extra multicast-specific attributes that can be controlled. Our next example implements the multicast version of `SendUDP.java` (see page 57).

SendUDPMulticast.c

```
 0  import java.net.*;  // for MulticastSocket, DatagramPacket, and InetAddress
 1  import java.io.*;   // for IOException
 2
 3  public class SendUDPMulticast {
 4
 5    public static void main(String args[]) throws Exception {
 6
 7      if ((args.length < 2) || (args.length > 3)) // Test for correct # of args
 8        throw new IllegalArgumentException(
 9          "Parameter(s): <Multicast Addr> <Port> [<TTL>]");
10
11      InetAddress destAddr = InetAddress.getByName(args[0]);  // Destination address
12      if (!destAddr.isMulticastAddress())  // Test if multicast address
13        throw new IllegalArgumentException("Not a multicast address");
14
15      int destPort = Integer.parseInt(args[1]);  // Destination port
```

```
16
17      int TTL;   // Time To Live for datagram
18      if (args.length == 3)
19        TTL = Integer.parseInt(args[2]);
20      else
21        TTL = 1;  // Default TTL
22
23      ItemQuote quote = new ItemQuote(1234567890987654L, "5mm Super Widgets",
24                                      1000, 12999, true, false);
25
26      MulticastSocket sock = new MulticastSocket(); // Multicast socket to sending
27      sock.setTimeToLive(TTL);                      // Set TTL for all datagrams
28
29      ItemQuoteEncoder encoder = new ItemQuoteEncoderText();  // Text encoding
30      byte[] codedQuote = encoder.encode(quote);
31
32      // Create and send a datagram
33      DatagramPacket message = new DatagramPacket(codedQuote, codedQuote.length,
34                                                  destAddr, destPort);
35      sock.send(message);
36
37      sock.close();
38    }
39  }
```

SendUDPMulticast.c

The only significant differences between our unicast and multicast senders are that 1) we verify that the given address is multicast, and 2) we set the initial Time To Live (TTL) value for the multicast datagram. Every IP datagram contains a TTL, initialized to some default value and decremented (usually by one) by each router that forwards the packet. When the TTL reaches zero, the packet is discarded. By setting the initial value of the TTL, we limit the distance a packet can travel from the sender.[2]

Unlike broadcast, network multicast duplicates the message only to a specific set of receivers. This set of receivers, called a *multicast group*, is identified by a shared multicast (or group) address. Receivers need some mechanism to notify the network of their interest in receiving data sent to a particular multicast address, so that the network can forward packets to them. This notification, called *joining a group*, is accomplished with the `joinGroup()` method of `MulticastSocket`. Our multicast receiver joins a specified group, receives and prints a single multicast message from that group, and exits.

[2] The rules for multicast TTL are actually not quite so simple. It is not necessarily the case that a packet with TTL = 4 can travel four hops from the sender; however, it will not travel *more* than four hops.

RecvUDPMulticast.java

```
 0  import java.net.*;  // for MulticastSocket, DatagramPacket, and InetAddress
 1  import java.io.*;   // for IOException
 2
 3  public class RecvUDPMulticast implements ItemQuoteTextConst {
 4
 5    public static void main(String[] args) throws Exception {
 6
 7      if (args.length != 2)  // Test for correct # of args
 8        throw new IllegalArgumentException("Parameter(s): <Multicast Addr> <Port>");
 9
10      InetAddress address = InetAddress.getByName(args[0]);  // Multicast address
11      if (!address.isMulticastAddress())  // Test if multicast address
12        throw new IllegalArgumentException("Not a multicast address");
13
14      int port = Integer.parseInt(args[1]);  // Multicast port
15
16      MulticastSocket sock = new MulticastSocket(port); // Multicast receiving socket
17      sock.joinGroup(address);                          // Join the multicast group
18
19      // Create and receive a datagram
20      DatagramPacket packet = new DatagramPacket(
21        new byte[MAX_WIRE_LENGTH], MAX_WIRE_LENGTH);
22      sock.receive(packet);
23
24      ItemQuoteDecoder decoder = new ItemQuoteDecoderText();  // Text decoding
25      ItemQuote quote = decoder.decode(packet);
26      System.out.println(quote);
27
28      sock.close();
29    }
30  }
```

RecvUDPMulticast.java

The only significant difference between our multicast and unicast receiver is that the multicast receiver must join the multicast group by supplying the desired multicast address. The book's Web site also contains another example of a sender and receiver multicast pair. `MulticastImageSender.java` transmits a set of images (JPEG or GIF) specified on the command line, in three-second intervals. `MulticastImageReceiver.java` receives each image and displays it in a window.

Multicast datagrams can, in fact, be sent from a `DatagramSocket` by simply using a multicast address. You can test this by using `SendUDP.java` (see page 57) to send to the multicast receiver. However, a `MulticastSocket` has a few capabilities that a `DatagramSocket` does not,

including 1) allowing specification of the datagram TTL, and 2) allowing the interface through which datagrams are sent to the group to be specified/changed (an interface is identified by its Internet address). A multicast receiver, on the other hand, *must* use a `MulticastSocket` because it needs the ability to join a group.

`MulticastSocket` is a subclass of `DatagramSocket`, so it provides all of the `DatagramSocket` methods. We only present methods specific to or adapted for `MulticastSocket`.

MulticastSocket

Constructors

`MulticastSocket()`

`MulticastSocket(`**int** *localPort*`)`

Constructs a datagram socket that can perform some additional multicast operations. The second form of the constructor specifies the local port. If the local port is not specified, the socket is bound to any available local port.

localPort	Local port. A localPort of 0 allows the constructor to pick any available port.

Operators

void `joinGroup(`**InetAddress** *groupAddress*`)`

void `leaveGroup(`**InetAddress** *groupAddress*`)`

Join/leave a multicast group. A socket may be a member of multiple groups simultaneously. Joining a group of which this socket is already a member or leaving a group of which this socket is not a member may generate an exception.

groupAddress	Multicast address identifying group

void `send(`**DatagramPacket** *packet*, **byte** *TTL*`)`

Send a datagram from this socket with the specified TTL.

packet	Packet to transmit. Either the packet must specify a destination address or the UDP socket must have a specified remote address and port (see `connect()`).
TTL	Time to live for this packet

Accessors

InetAddress `getInterface()`

void `setInterface(`**InetAddress** *interface*`)`

Returns/sets the interface to use for multicast operations on this socket. This is primarily used on hosts with multiple interfaces. Join/leave requests and datagrams

will be sent and datagrams will be received using this interface. The default multicast interface is platform dependent.

interface Address of one of host's multicast interfaces

int `getTimeToLive()`

void `setTimeToLive(`**int** *TTL*`)`

Returns/sets the Time To Live for all datagrams sent on this socket. This can be overridden on a per-datagram basis using the `send()` method that takes the TTL as a parameter.

TTL Time To Live for this packet

The decision to use broadcast or multicast depends on several factors, including the network location of receivers and the knowledge of the communicating parties. The scope of a broadcast on the Internet is restricted to a local broadcast network, placing severe restrictions on the location of the broadcast receivers. Multicast communication may include receivers anywhere in the network,[3] so multicast has the advantage that it can cover a distributed set of receivers. The disadvantage of IP multicast is that receivers must know the address of a multicast group to join. Knowledge of an address is not required to receive broadcast. In some contexts, this makes broadcast a better mechanism than multicast for discovery. All hosts can receive broadcast by default, so it is simple to ask all hosts on a single network a question like "Where's the printer?"

UDP unicast, multicast, and broadcast are all implemented using an underlying UDP socket. The semantics of most implementations are such that a UDP datagram will be delivered to all sockets bound to the destination port of the packet. That is, a `DatagramSocket` or `MulticastSocket` instance bound to a local port X (with local address not specified, i.e., a wild card), on a host with address Y will receive any UDP datagram destined for port X that is 1) unicast with destination address Y, 2) multicast to a group that *any* application on Y has joined, or 3) broadcast where it can reach host Y. A receiver can use `connect()` to limit the datagram source address and port. Also, a `DatagramSocket` can specify the local unicast address, which prevents delivery of multicast and broadcast packets. See `UDPEchoClientTimeout.java` for an example of destination address verification and Section 5.5 for details on datagram demultiplexing.

4.4 Socket Options

The TCP/IP protocol developers spent a good deal of time thinking about the default behaviors that would satisfy most applications. (If you doubt this, read RFCs 1122 and 1123, which describe in excruciating detail the recommended behaviors—based on years of experience—

[3] At the time of writing of this book, there are severe limitations on who can receive multicast traffic on the Internet; however, multicast availability should improve over time. Multicast should work if the sender and receivers are on the same LAN.

for implementations of the TCP/IP protocols.) For most applications, the designers did a good job; however, it is seldom the case that "one size fits all" really fits all. We have already seen an example in our UDP echo client. By default, the `receive()` method of `DatagramSocket` blocks indefinitely waiting on a datagram. In our example, we change that behavior by specifying a timeout for receives on the UDP socket using `setSoTimeout()`. In socket parlance, each type of behavior we can change is called a *socket option*. In Java, the socket type (e.g., `Socket`, `ServerSocket`, `DatagramSocket`, and `MulticastSocket`) determines the applicable socket options, which are typically queried and controlled using accessor methods like `getSoTimeout()` and `setSoTimeout()`. Unfortunately, the Java API allows access to only a subset of the options in the underlying sockets API. This is at least partly because options tend to vary in availability from platform to platform, and Java is all about portability. However, as the versions wear on, access to more and more socket options is being added in Java. Check the latest official documentation for the various socket types to see the available options.

4.5 Closing Connections

You've probably never given much thought to who closes a connection. In phone conversations, either side can start the process of terminating the call. It typically goes something like this:

"Well, I guess I'd better go."
"OK. Bye."
"Bye."

Network protocols, on the other hand, are typically very specific about who "closes" first. In the echo protocol, Figure 4.1(a), the server dutifully echoes everything the client sends. When the client is finished, it calls `close()`. After the server has received and echoed all of the data sent before the client's call to `close()`, its read operation returns a −1, indicating that the client is finished. The server then calls `close()` on its socket. The close is a critical part of the protocol because without it the server doesn't know when the client is finished sending characters to echo. In HTTP, Figure 4.1(b), it's the server that initiates the connection close. Here, the client sends a request ("GET") to the server, and the server responds by sending a header (normally

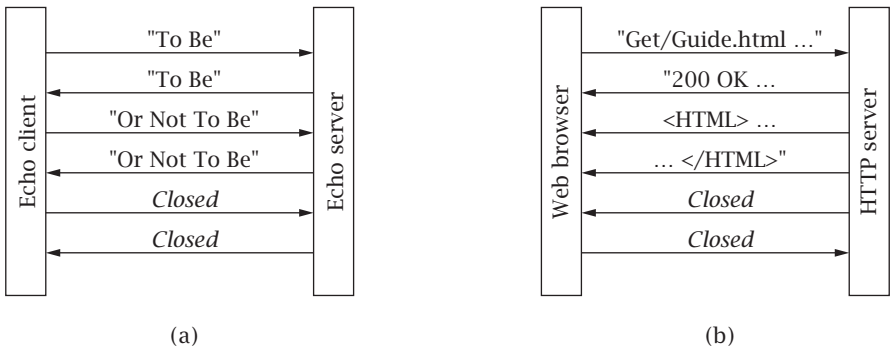


Figure 4.1: Echo (a) and HTTP (b) protocol termination.

starting with "200 OK"), followed by the requested file. Since the client does not know the size of the file, the server must indicate the end-of-file by closing the socket.

Calling `close()` on a `Socket` terminates *both* directions (input and output) of data flow. (Subsection 5.4.2 provides a more detailed description of TCP connection termination.) Once an endpoint (client or server) closes the socket, it can no longer send *or receive* data. This means that `close()` can only be used to signal the other end when the caller is completely finished communicating. In the echo protocol, once the server receives the close from the client, it immediately closes. In effect, the client close indicates that the communication is completed. HTTP works the same way, except that the server is the terminator.

Let's consider a different protocol. Suppose you want a compression server that takes a stream of bytes, compresses them, and sends the compressed stream back to the client. Which endpoint should close the connection? Since the stream of bytes from the client is arbitrarily long, the client needs to close the connection so that the server knows when the stream of bytes to be compressed ends. When should the client call `close()`? If the client calls `close()` on the socket immediately after it sends the last byte of data, it will not be able to receive the last bytes of compressed data. Perhaps the client could wait until it receives all of the compressed data before it closes, as the echo protocol does. Unfortunately, neither the server nor the client knows how many bytes to expect, so this will not work either. What is needed is a way to tell the other end of the connection "I am through sending," without losing the ability to receive.

Fortunately, sockets provide a way to do this. The `shutdownInput()` and `shutdownOutput()` methods of `Socket` allow the I/O streams to be closed independently. After `shutdownInput()`, the socket's input stream can no longer be used. Any undelivered data is silently discarded, and any attempt to read from the socket's input stream will return −1. After `shutdownOutput()` is called on a `Socket`, no more data may be sent on the socket's output stream. Attempts to write to the stream throw an `IOException`. Any data written before the call to `shutdownOutput()` may be read by the remote socket. After this, a read on the input stream of the remote socket will return −1. An application calling `shutdownOutput()` can continue to read from the socket and, similarly, data can be written after calling `shutdownInput()`.

In the compression protocol (see Figure 4.2), the client writes the bytes to be compressed, closing the output stream using `shutdownOutput()` when finished sending, and reads the compressed byte stream from the server. The server repeatedly reads the uncompressed data

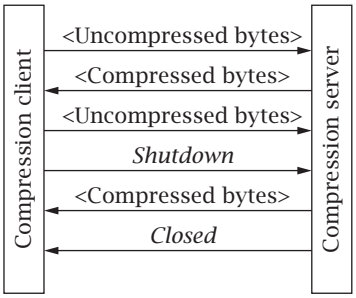


Figure 4.2: Compression protocol termination.

and writes the compressed data until the client performs a shutdown, causing the server read to return −1, indicating an end-of-stream. The server then closes the connection and exits. After the client calls `shutdownOutput()`, it needs to read any remaining compressed bytes from the server.

Our compression client, `CompressClient.java`, implements the client side of the compression protocol. The uncompressed bytes are read from the file specified on the command line, and the compressed bytes are written to a new file. If the uncompressed filename is "`data`", the compressed filename is "`data.gz`". Note that this implementation works for small files, but that there is a flaw that causes deadlock for large files. (We discuss and correct this shortcoming in Section 5.2.)

CompressClient.java

```
 0  import java.net.*;  // for Socket
 1  import java.io.*;   // for IOException and [File]Input/OutputStream
 2
 3  public class CompressClient {
 4
 5    public static final int BUFSIZE = 256;  // Size of read buffer
 6
 7    public static void main(String[] args) throws IOException {
 8
 9      if (args.length != 3)  // Test for correct # of args
10        throw new IllegalArgumentException("Parameter(s): <Server> <Port> <File>");
11
12      String server = args[0];               // Server name or IP address
13      int port = Integer.parseInt(args[1]);  // Server port
14      String filename = args[2];             // File to read data from
15
16      // Open input and output file (named input.gz)
17      FileInputStream fileIn = new FileInputStream(filename);
18      FileOutputStream fileOut = new FileOutputStream(filename + ".gz");
19
20      // Create socket connected to server on specified port
21      Socket sock = new Socket(server, port);
22
23      // Send uncompressed byte stream to server
24      sendBytes(sock, fileIn);
25
26      // Receive compressed byte stream from server
27      InputStream sockIn = sock.getInputStream();
28      int bytesRead;                      // Number of bytes read
29      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
30      while ((bytesRead = sockIn.read(buffer)) != -1) {
31        fileOut.write(buffer, 0, bytesRead);
```

```
32        System.out.print("R");   // Reading progress indicator
33      }
34      System.out.println();      // End progress indicator line
35
36      sock.close();     // Close the socket and its streams
37      fileIn.close();   // Close file streams
38      fileOut.close();
39    }
40
41    private static void sendBytes(Socket sock, InputStream fileIn)
42        throws IOException {
43      OutputStream sockOut = sock.getOutputStream();
44      int bytesRead;                      // Number of bytes read
45      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
46      while ((bytesRead = fileIn.read(buffer)) != -1) {
47        sockOut.write(buffer, 0, bytesRead);
48        System.out.print("W");   // Writing progress indicator
49      }
50      sock.shutdownOutput();     // Finished sending
51    }
52  }
```

CompressClient.java

1. **Application setup and parameter parsing:** lines 0–14
2. **Create socket and open files:** lines 16–21
3. **Invoke** `sendBytes()` **to transmit bytes:** lines 23–24
4. **Receive the compressed data stream:** lines 26–34
   The while loop receives the compressed data stream and writes the bytes to the output file until an end-of-stream is signaled by a −1 from `read()`.
5. **Close socket and file streams:** lines 36–38
6. `sendBytes()`**:** lines 41–51
   Given a socket connected to a compression server and the file input stream, read all of the uncompressed bytes from the file and write them to the socket output stream.
   - **Get socket output stream:** line 43
   - **Send uncompressed bytes to compression server:** lines 44–49
     The while loop reads from the input stream (in this case from a file) and repeats the bytes to the socket output stream until end-of-file, indicated by −1 from `read()`. Each write is indicated by a “W” printed to the console.

- **Shut down the socket output stream:** line 50
  After reading and sending all of the bytes from the input file, shut down the output stream, notifying the server that the client is finished sending. The close will cause a −1 return from `read()` on the server.

To implement the compression server, we simply write a protocol and factory for our threaded server architecture. Our protocol implementation, `CompressProtocolFactory.java`, implements the server-side compression protocol using the GZIP compression algorithm. The server receives the uncompressed bytes from the client and writes them to a `GZIPOutputStream`, which wraps the socket's output stream.

CompressProtocolFactory.java

```
 0  import java.net.*;       // for Socket
 1  import java.io.*;        // for IOException and Input/OutputStream
 2  import java.util.*;      // for ArrayList
 3  import java.util.zip.*;  // for GZIPOutputStream
 4
 5  public class CompressProtocolFactory implements ProtocolFactory {
 6
 7    public static final int BUFSIZE = 1024;   // Size of receive buffer
 8
 9    public Runnable createProtocol(final Socket clntSock, final Logger logger) {
10      return new Runnable() {
11        public void run() {
12          CompressProtocolFactory.handleClient(clntSock, logger);
13        }
14      };
15    }
16
17    public static void handleClient(Socket clntSock, Logger logger) {
18      ArrayList entry = new ArrayList();
19      entry.add("Client address and port = " +
20        clntSock.getInetAddress().getHostAddress() + ":" +
21        clntSock.getPort());
22      entry.add("Thread = " + Thread.currentThread().getName());
23
24      try {
25        // Get the input and output streams from socket
26        InputStream in = clntSock.getInputStream();
27        GZIPOutputStream out = new GZIPOutputStream(clntSock.getOutputStream());
28
29        byte[] buffer = new byte[BUFSIZE];   // Allocate read/write buffer
30        int bytesRead;                       // Number of bytes read
31        // Receive until client closes connection, indicated by -1 return
```

```
32        while ((bytesRead = in.read(buffer)) != -1)
33          out.write(buffer, 0, bytesRead);
34
35        out.finish();      // Flush bytes from GZIPOutputStream
36      } catch (IOException e) {
37        logger.writeEntry("Exception = " +  e.getMessage());
38      }
39
40      try {  // Close socket
41        clntSock.close();
42      } catch (IOException e) {
43        entry.add("Exception = " +  e.getMessage());
44      }
45
46      logger.writeEntry(entry);
47    }
48  }
```

CompressProtocolFactory.java

1. **Factory method for compression protocol:** lines 9–15
   `createProtocol()` returns an anonymous class instance that implements the `Runnable` interface. The `run()` method of this instance simply calls the static method `CompressProtocolFactory.handleClient()`, which implements the server-side compression protocol. Note that we do *not* need a separate `CompressProtocol` class, because `createProtocol()` returns the type of instance (one that implements `Runnable`) that we need.
2. `handleClient()`: lines 17–38
   Given a socket connected to the compression client, read the uncompressed bytes from the client and write the compressed bytes back.
   - **Get socket I/O streams:** lines 26–27
     The socket's output stream is wrapped in a `GZIPOutputStream`. The sequence of bytes written to this stream is compressed, using the GZIP algorithm, before being written to the underlying output stream.
   - **Read uncompressed and write compressed bytes:** lines 29–33
     The while loop reads from the socket input stream and writes to the `GZIPOutputStream`, which in turn writes to the socket output stream, until the end-of-stream indication is received.
   - **Flush and close:** lines 35–44
     Calling `finish()` on the `GZIPOutputStream` is necessary to flush any bytes that may be buffered by the compression algorithm.

A simple iterative version of the server can be found in `CompressServer.java` on the book's Web site.

4.6 Applets

Applets can perform network communication using TCP/IP sockets, but there are severe restrictions on how and with whom they can converse. Without such restrictions, unsuspecting Web browsers might execute malicious applets that could, for example, send fake email, attempt to hack other systems while the browser user gets the blame, and so on. These security restrictions are enforced by the Java security manager, and violations by the applet result in a `SecurityException`. Typically, browsers only allow applets to communicate with the host that served the applet. This means that applets are usually restricted to communicating with applications executing on that host, usually a Web server originating the applet. The list of security restrictions and general applet programming is beyond the scope of this book. It is worth noting, however, that the default security restrictions can be altered, if allowed by the browser user.

Suppose that you wanted to implement an applet that allowed users to type and save notes to themselves on their browser. Browser security restrictions prevent applets from saving data directly on the local file system, so you would need some other means besides local disk I/O to save the notes. `FileClientApplet.java` (available from the book's Web site) is an applet that allows the user to type text into an editor window and, by clicking the "Save" button, copy the text over the network to a server (running on port 5000). The server, `TCPFileServer.java` (also on the book's Web site), saves the data to a file. It takes a port (use 5000 to work with the applet) and the name of the file. The server must execute on the Web server that serves the applet to the browser. Note that there is nothing applet specific about the server. `FileClientApplet.html` on the Web site demonstrates how to integrate the applet into a Web page. Be warned that the applet is based on Swing, and most browsers don't have the Swing library. The HTML file should download the necessary file to make this work, but it is not guaranteed.

4.7 Wrapping Up

We have discussed some of the ways Java provides access to advanced features of the sockets API, and how built-in features such as threads can be used with socket programs. In addition to these facilities, Java provides several mechanisms that operate on top of TCP or UDP and attempt to hide the complexity of protocol development. For example, Java Remote Method Invocation (RMI) allows Java objects on different hosts to invoke one another's methods as if the objects all reside locally. The URL class and associated classes provide a framework for developing Web-related programs. Many other standard Java library mechanisms exist, providing an amazing range of services. These mechanisms are beyond the scope of this book; however, we encourage you to look at the book's Web site for descriptions and code examples for some of these libraries.

4.8 Exercises

1. State precisely the conditions under which an iterative server is preferable to a multiprocessing server.

2. Would you ever need to implement a timeout in a client or server that uses TCP?
3. How can you determine the minimum and maximum allowable sizes for a socket's send and receive buffers? Determine the minimums for your system.
4. Write an iterative dispatcher using the dispatching framework from this chapter.
5. Write the server side of a random-number server using the protocol factory framework from this chapter. The client will connect and send the upper bound, *B*, on the random number to the server. The server should return a random number between 1 and *B*, inclusive. All numbers should be specified in binary format as 4-byte, two's-complement, big-endian integers.
6. Modify `TCPEchoClient.java` so that it closes its output side of the connection before attempting to receive any echoed data.

chapter 5

Under the Hood

Some of the subtleties of network programming are difficult to grasp without some understanding of the data structures associated with the socket implementation and certain details of how the underlying protocols work. This is especially true of TCP sockets (i.e., instances of `Socket`). This chapter describes some of what goes on under the hood when you create and use an instance of `Socket` or `ServerSocket`. (The initial discussion and Section 5.5 apply as well to `DatagramSocket` and `MulticastSocket`. However, most of this chapter focuses on TCP sockets, that is, `Socket` and `ServerSocket`.) Please note that this description covers only the normal sequence of events and glosses over many details. Nevertheless, we believe that even this basic level of understanding is helpful. Readers who want the full story are referred to the TCP specification [13] or to one of the more comprehensive treatises on the subject [3, 22].

Figure 5.1 is a simplified view of some of the information associated with a `Socket` instance. The classes are supported by an underlying implementation that is provided by the JVM and/or the platform on which it is running (i.e., the "socket layer" of the host's OS). Operations on the Java objects are translated into manipulations of this underlying abstraction. In this chapter, "`Socket`" refers generically to one of the classes in Figure 5.1, while "socket" refers to the underlying abstraction, whether it is provided by an underlying OS or the JVM implementation itself (e.g., in an embedded system). It is important to note that other (possibly non-Java) programs running on the same host may be using the network via the underlying socket abstraction, and thus competing with Java `Socket` instances for resources such as ports.

By "socket structure" here we mean the collection of data structures in the underlying implementation (of both the JVM and TCP/IP, but primarily the latter) that contain the information associated with a particular `Socket` instance. For example, the socket structure contains, among other information

- The local and remote Internet addresses and port numbers associated with the socket. The local Internet address (labeled "Local IP" in the figure) is one of those assigned to the local host; the local port is set at `Socket` creation time. The remote address and port identify the remote socket, if any, to which the local socket is connected. We will say more

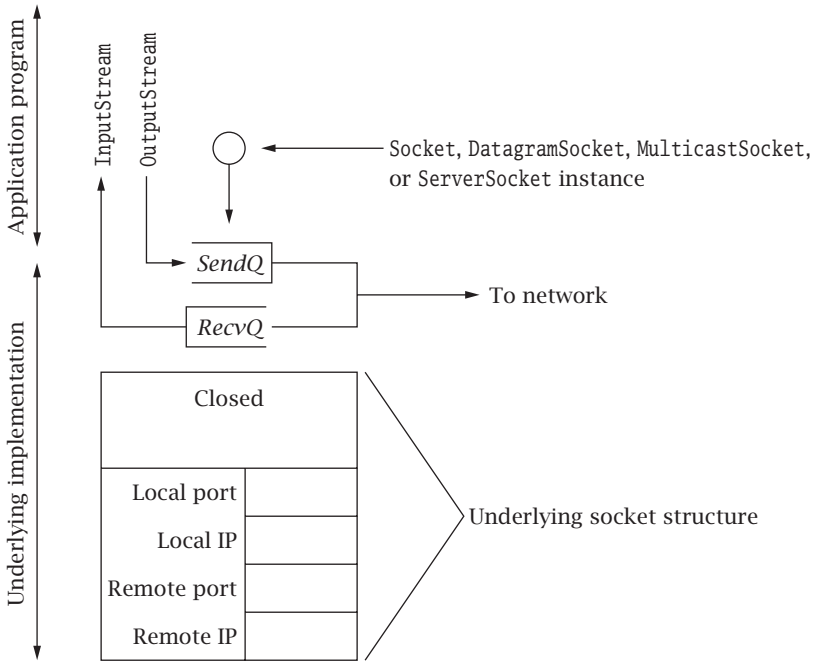


Figure 5.1: Data structures associated with a socket.

about how and when these values are determined shortly (Section 5.5 contains a concise summary).

- A FIFO queue of received data waiting to be delivered and a queue for data waiting to be transmitted.
- For a TCP socket, additional protocol state information relevant to the opening and closing TCP handshakes. In Figure 5.1, the state is "Closed"; all sockets start out in the Closed state.

Knowing that these data structures exist and how they are affected by the underlying protocols is useful because they control various aspects of the behavior of the various `Socket` objects. For example, because TCP provides a *reliable* byte-stream service, a copy of any data written to a `Socket`'s `OutputStream` must be kept until it has been successfully received at the other end of the connection. Writing data to the output stream does *not* imply that the data has actually been sent—only that it has been copied into the local buffer. Even `flush()`ing a `Socket`'s `OutputStream` doesn't guarantee that anything goes over the wire immediately. Moreover, the nature of the byte-stream service means that message boundaries are *not* preserved in the input stream. As we saw in Section 3.3, this complicates the process of receiving and parsing for some protocols. On the other hand, with a `DatagramSocket`, packets are *not* buffered for retransmission, and by the time a call to the `send()` method returns, the data has been given to

the network subsystem for transmission. If the network subsystem cannot handle the message for some reason, the packet is silently dropped (but this is rare).

The next three sections deal with some of the subtleties of sending and receiving with TCP's byte-stream service. Then, Section 5.4 considers the connection establishment and termination of the TCP protocol. Finally, Section 5.5 discusses the process of matching incoming packets to sockets and the rules about binding to port numbers.

5.1 Buffering and TCP

As a programmer, the most important thing to remember when using a TCP socket is this:

> *You cannot assume any correspondence between writes to the output stream at one end of the connection and reads from the input stream at the other end.*

In particular, data passed in a single invocation of the output stream's `write()` method at the sender can be spread across multiple invocations of the input stream's `read()` method at the other end; and a single `read()` may return data passed in multiple `write()`s. To see this, consider a program that does the following:

```
byte[] buffer0 = new byte[1000];
byte[] buffer1 = new byte[2000];
byte[] buffer2 = new byte[5000];
.
.
.
Socket s = new Socket(destAddr, destPort);
OutputStream out = s.getOutputStream();
.
.
.
out.write(buffer0);
.
.
.
out.write(buffer1);
.
.
.
out.write(buffer2);
.
.
.
s.close();
```

where the ellipses represent code that sets up the data in the buffers but contains no other calls to `out.write()`. Throughout this discussion, "in" refers to the `InputStream` of the receiver's `Socket`, and "out" refers to the `OutputStream` of the sender's `Socket`.

This TCP connection transfers 8000 bytes to the receiver. The way these 8000 bytes are grouped for delivery at the receiving end of the connection depends on the timing between the `out.write()`s and `in.read()`s at the two ends of the connection—as well as the size of the buffers provided to the `in.read()` calls.

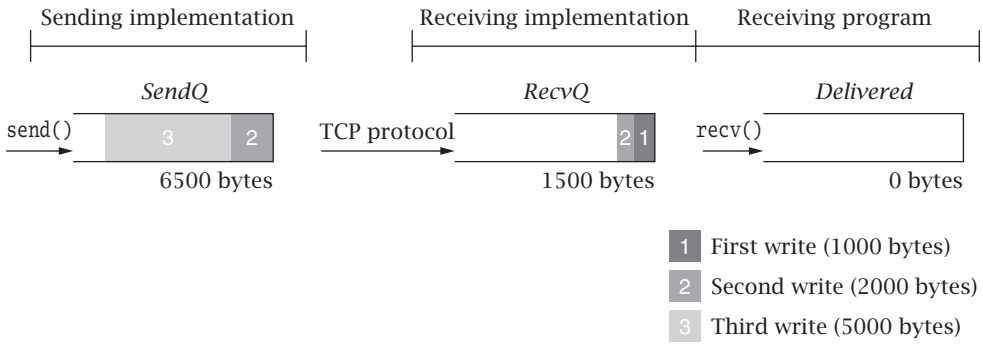


Figure 5.2: State of the three queues after three writes.

We can think of the sequence of all bytes sent (in one direction) on a TCP connection up to a particular instant in time as being divided into three FIFO queues:

1. *SendQ*: Bytes buffered in the underlying implementation at the sender that have been written to the output stream but not yet successfully transmitted to the receiving host.
2. *RecvQ*: Bytes buffered in the underlying implementation at the receiver waiting to be delivered to the receiving program—that is, read from the input stream.
3. *Delivered*: Bytes already read from the input stream by the receiver.

A call to `out.write()` at the sender appends bytes to *SendQ*. The TCP protocol is responsible for moving bytes—in order—from *SendQ* to *RecvQ*. It is important to realize that this transfer cannot be controlled or directly observed by the user program, and that it occurs in chunks whose sizes are more or less independent of the size of the buffers passed in `write()`s. Bytes are moved from *RecvQ* to *Delivered* as they are read from the `Socket`'s `InputStream` by the receiving program; the size of the transferred chunks depends on the amount of data in *RecvQ* and the size of the buffer given to `read()`.

Figure 5.2 shows one possible state of the three queues *after* the three `out.write()`s in the example above, but *before* any `in.reads()`s at the other end. The different shading patterns denote bytes passed in the three different invocations of `write()` shown above.

Now suppose the receiver calls `read()` with a byte array of size 2000. The `read()` call will move all of the 1500 bytes present in the waiting-for-delivery (*RecvQ*) queue into the byte array and return the value 1500. Note that this data includes bytes passed in both the first and second calls to `write()`. At some time later, after TCP has completed transfer of more data, the three partitions might be in the state shown in Figure 5.3.

If the receiver now calls `read()` with a buffer of size 4000, that many bytes will be moved from the waiting-for-delivery (*RecvQ*) queue to the already-delivered (*Delivered*) queue; this includes the remaining 1500 bytes from the second `write()`, plus the first 2500 bytes from the third `write()`. The resulting state of the queues is shown in Figure 5.4.

The number of bytes returned by the next call to `read()` depends on the size of the buffer and the timing of the transfer of data over the network from the send-side socket/TCP

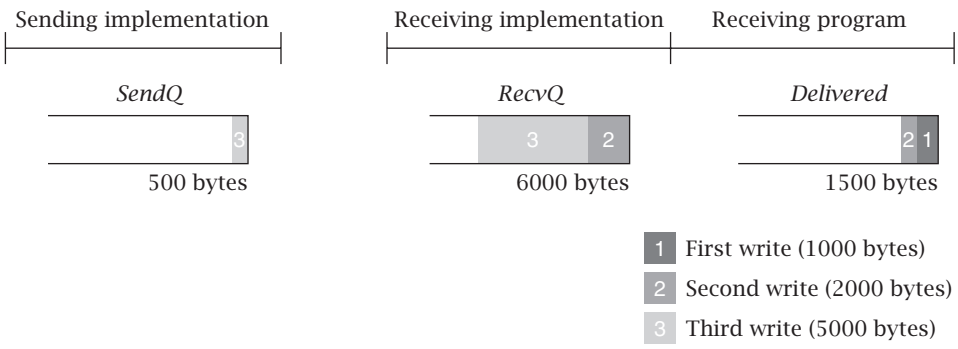


Figure 5.3: After first `read()`.

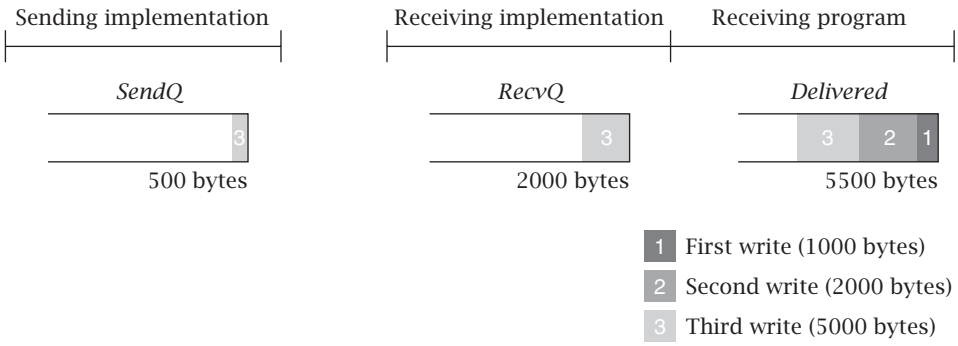


Figure 5.4: After another `read()`.

implementation to the receive-side implementation. The movement of data from the *SendQ* to the *RecvQ* buffer has important implications for the design of application protocols. We have already encountered the need to parse messages as they are received via a `Socket` when in-band delimiters are used for framing (see Section 3.3). In the following sections, we consider two more subtle ramifications.

5.2 Buffer Deadlock

Application protocols have to be designed with some care to avoid *deadlock*—that is, a state in which each peer is blocked waiting for the other to do something. For example, it is pretty obvious that if both client and server try to receive immediately after a connection is established, deadlock will result. Deadlock can also occur in less immediate ways.

The buffers *SendQ* and *RecvQ* in the implementation have limits on their capacity. Although the actual amount of memory they use may grow and shrink dynamically, a hard limit is necessary to prevent all of the system's memory from being gobbled up by a single

TCP connection under control of a misbehaving program. Because these buffers are finite, they can fill up, and it is this fact, coupled with TCP's *flow control* mechanism, that leads to the possibility of another form of deadlock.

Once *RecvQ* is full, the TCP flow control mechanism kicks in and prevents the transfer of any bytes from the sending host's *SendQ,* until space becomes available in *RecvQ* as a result of the receiver calling the input stream's `read()` method. (The purpose of the flow control mechanism is to ensure that the sender does not transmit more data than the receiving system can handle.) A sending program can continue to call send until *SendQ* is full; however, once *SendQ* is full, a call to `out.write()` will block until space becomes available, that is, until some bytes are transferred to the receiving socket's *RecvQ*. If *RecvQ* is also full, everything stops until the receiving program calls `in.read()` and some bytes are transferred to *Delivered*.

Let's assume the sizes of *SendQ* and *RecvQ* are *SQS* and *RQS*, respectively. A `write()` call with a byte array of size n such that $n > SQS$ will not return until at least $n - SQS$ bytes have been transferred to *RecvQ* at the receiving host. If n exceeds $(SQS + RQS)$, `write()` cannot return until after the receiving program has read at least $n - (SQS + RQS)$ bytes from the input stream. If the receiving program does not call `read()`, a large `send()` may not complete successfully. In particular, if both ends of the connection invoke their respective output streams' `write()` method simultaneously with buffers greater than $SQS + RQS$, deadlock will result: neither write will ever complete, and both programs will remain blocked forever.

As a concrete example, consider a connection between a program on Host A and a program on Host B. Assume *SQS* and *RQS* are 500 at both A and B. Figure 5.5 shows what happens when both programs try to send 1500 bytes at the same time. The first 500 bytes of data in the program at Host A have been transferred to the other end; another 500 bytes have been copied into *SendQ* at Host A. The remaining 500 bytes cannot be sent—and therefore `out.write()` will not return—until space frees up in *RecvQ* at Host B. Unfortunately, the same situation holds in the program at Host B. Therefore, neither program's `write()` call will ever complete.

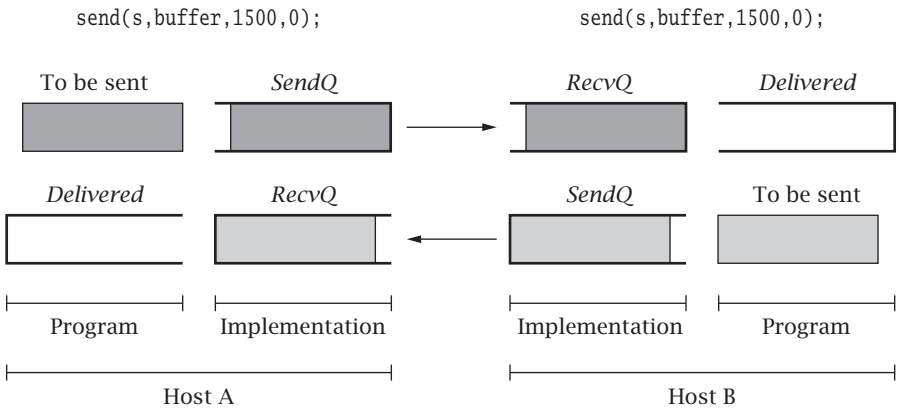


Figure 5.5: Deadlock due to simultaneous `write()`s to output streams at opposite ends of the connection.

The moral of the story: Design the protocol carefully to avoid sending large quantities of data simultaneously in both directions.

Can this really happen? Let's review the compression protocol example in Section 4.5. Try running the compression client with a large file that is still large *after compression*. The precise definition of "large" here depends on your system, but a file that is already compressed and exceeds 2MB should do nicely. For each read/write, the compression client prints an "R"/"W" to the console. If both the uncompressed and compressed versions of the file are large enough, your client will print a series of Ws and then stop without terminating or printing any Rs.

Why does this happen? The program `CompressClient.java` sends *all* of the uncompressed data to the compression server *before* it attempts to read anything from the compressed stream. The server, on the other hand, simply reads the uncompressed byte sequence and writes the compressed sequence back to the client. (The number of bytes the server reads before it writes some compressed data depends on the compression algorithm it uses.) Consider the case where *SendQ* and *RecvQ* for both client and server hold 500 bytes each and the client sends a 10,000-byte (uncompressed) file. Suppose also that for this file the server reads about 1000 bytes and then writes 500 bytes, for a 2:1 compression ratio. After the client sends 2000 bytes, the server will eventually have read them all and sent back 1000 bytes, and the client's *RecvQ* and the server's *SendQ* will both be full. After the client sends another 1000 bytes and the server reads them, the server's subsequent attempt to write will block. When the client sends the next 1000 bytes, the client's *SendQ* and the server's *RecvQ* will both fill up. The next client write will block, creating deadlock.

How do we solve this problem? The easiest solution is to execute the client writing and reading loop in separate threads. One thread repeatedly reads a buffer of uncompressed bytes from a file and sends them to the server until the end of the file is reached, whereupon it calls `shutdownOutput()` on the socket. The other thread repeatedly reads a buffer of compressed bytes from the server and writes them to the output file, until the input stream ends (i.e., the server closes the socket). When one thread blocks, the other thread can proceed independently. We can easily modify our client to follow this approach by putting the call to `SendBytes()` in `CompressClient.java` inside a thread as follows:

```
Thread thread = new Thread() {
  public void run() {
    try {
      SendBytes(sock, fileIn);
    } catch (Exception ignored) {}
  }
};
thread.start();
```

See `CompressClientNoDeadlock.java` on the book's Web site for the complete example. Can we solve this problem without using threads? To guarantee deadlock avoidance in a single threaded solution, we need nonblocking writes, which are not available in the current version of Java (see Section 4.2).

5.3 Performance Implications

The TCP implementation's need to copy user data into *SendQ* for potential retransmission also has implications for performance. In particular, the sizes of the *SendQ* and *RecvQ* buffers affect the throughput achievable over a TCP connection. Throughput refers to the rate at which bytes of user data from the sender are made available to the receiving program; in programs that transfer a large amount of data, we want to maximize this rate. In the absence of network capacity or other limitations, bigger buffers generally result in higher throughput.

The reason for this has to do with the cost of transferring data into and out of the buffers in the underlying implementation. If you want to transfer n bytes of data (where n is large), it is generally much more efficient to call `write()` once with a buffer of size n than it is to call it n times with a single byte.[1] However, if you call `write()` with a size parameter that is much larger than *SQS*, the system has to transfer the data from the user address space in *SQS*-sized chunks. That is, the socket implementation fills up the *SendQ* buffer, waits for data to be transferred out of it by the TCP protocol, refills *SendQ*, waits some more, and so on. Each time the socket implementation has to wait for data to be removed from *SendQ*, some time is wasted in the form of overhead (a context switch occurs). This overhead is comparable to that incurred by a completely new call to `write()`. Thus, the *effective* size of a call to `write()` is limited by the actual *SQS*. For reading from the `InputStream`, the same principle applies: however large the buffer we give to `read()`, it will be copied out in chunks no larger than *RQS*, with overhead incurred between chunks.

If you are writing a program for which throughput is an important performance metric, you will want to change the send and receive buffer sizes using the `setSendBufferSize()` and `setReceiveBufferSize()` methods of `Socket`. Although there is always a system-imposed maximum size for each buffer, it is typically significantly larger than the default on modern systems. Remember that these considerations apply only if your program needs to send an amount of data significantly larger than the buffer size, all at once. Note also that these factors may make little difference if the program deals with some higher-level stream derived from the `Socket`'s basic input stream (say, by using it to create an instance of `FilterOutputStream` or `PrintWriter`), which may perform its own internal buffering or add other overhead.

5.4 TCP Socket Life Cycle

When a new instance of the `Socket` class is created—either via one of the public constructors or by calling the `accept()` method of a `ServerSocket`—it can immediately be used for sending and receiving data. That is, when the instance is returned, it is already connected to a remote peer and the opening TCP message exchange, or handshake, has been completed by the implementation.

---

[1] The same thing generally applies to reading data from the `Socket`'s `InputStream`, although calling `read()` with a larger buffer does not guarantee that more data will be returned.

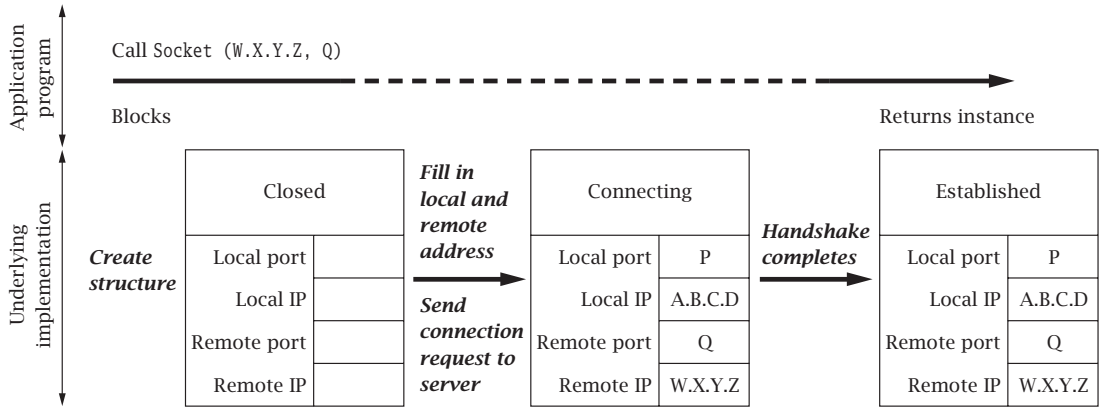


Figure 5.6: Client-side connection establishment.

Let us therefore consider in more detail how the underlying structure gets to and from the connected, or "Established," state; as you'll see later (see Section 5.4.2), these details affect the definition of reliability and the ability to create a `Socket` or `ServerSocket` bound to a particular port.

5.4.1 Connecting

The relationship between an invocation of the `Socket` constructor and the protocol events associated with connection establishment at the client are illustrated in Figure 5.6. In this and the remaining figures of this section, the large arrows depict external events that cause the underlying socket structures to change state. Events that occur in the application program—that is, method calls and returns—are shown in the upper part of the figure; events such as message arrivals are shown in the lower part of the figure. Time proceeds left to right in these figures. The client's Internet address is depicted as A.B.C.D, while the server's is W.X.Y.Z; the server's port number is Q.

When the client calls the `Socket` constructor with the server's Internet address, W.X.Y.Z, and port, Q, the underlying implementation creates a socket instance; it is initially in the Closed state. If the client did not specify the local address/port in the constructor call, a local port number (P), not already in use by another TCP socket, is chosen by the implementation. The local Internet address is also assigned; if not explicitly specified, the address of the network interface through which packets will be sent to the server is used. The implementation copies the local and remote addresses and ports into the underlying socket structure, and initiates the TCP connection establishment handshake.

The TCP opening handshake is known as a *3-way handshake* because it typically involves three messages: a connection request from client to server, an acknowledgment from server to client, and another acknowledgment from client back to server. The client TCP considers the connection to be established as soon as it receives the acknowledgment from the server.

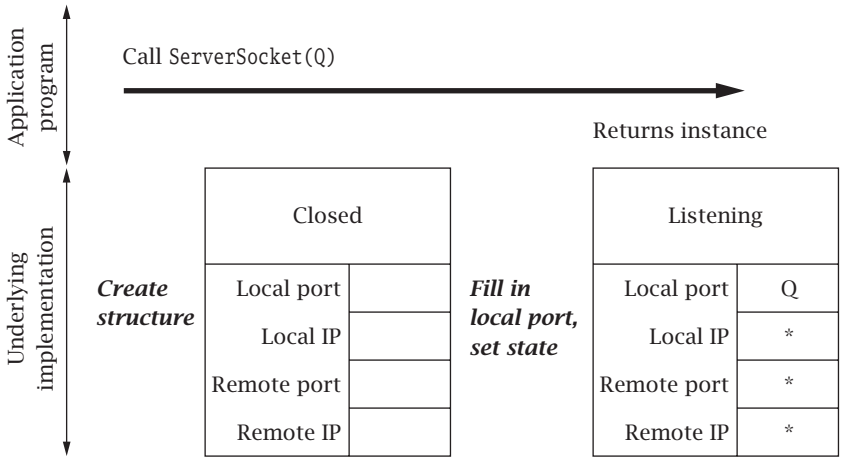


Figure 5.7: Server-side socket setup.

In the normal case, this happens quickly. However, the Internet is a best-effort network, and either the client's initial message or the server's response can get lost. For this reason, the TCP implementation retransmits handshake messages multiple times, at increasing intervals. If the client TCP does not receive a response from the server after some time, it *times out* and gives up. In this case the constructor throws an `IOException`. The connection timeout is generally long, and thus it can take on the order of minutes for a `Socket()` constructor to fail. If the server is not accepting connections—say, if there is no program associated with the given port at the destination—the server-side TCP will send a rejection message instead of an acknowledgment, and the constructor will throw an `IOException` almost immediately.

The sequence of events at the server side is rather different; we describe it in Figures 5.7, 5.8, and 5.9. The server first creates an instance of `ServerSocket` associated with its well-known port (here, Q). The socket implementation creates an underlying socket structure for the new `ServerSocket` instance, and fills in Q as the local port and the special *wildcard address* ("*" in the figures) for the local IP address. (The server may also specify a local IP address in the constructor, but typically it does not. In case the server host has more than one IP address, not specifying the local address allows the socket to receive connections addressed to any of the server host's addresses.) The state of the socket is set to "Listening", indicating that it is ready to accept incoming connection requests addressed to its port. This sequence is depicted in Figure 5.7.

The server can now call the `ServerSocket`'s `accept()` method, which blocks until the TCP opening handshake has been completed with some client and a new connection has been established. We therefore focus in Figure 5.8 on the events that occur in the TCP implementation when a client connection request arrives. Note that everything depicted in this figure happens "under the covers," in the TCP implementation.

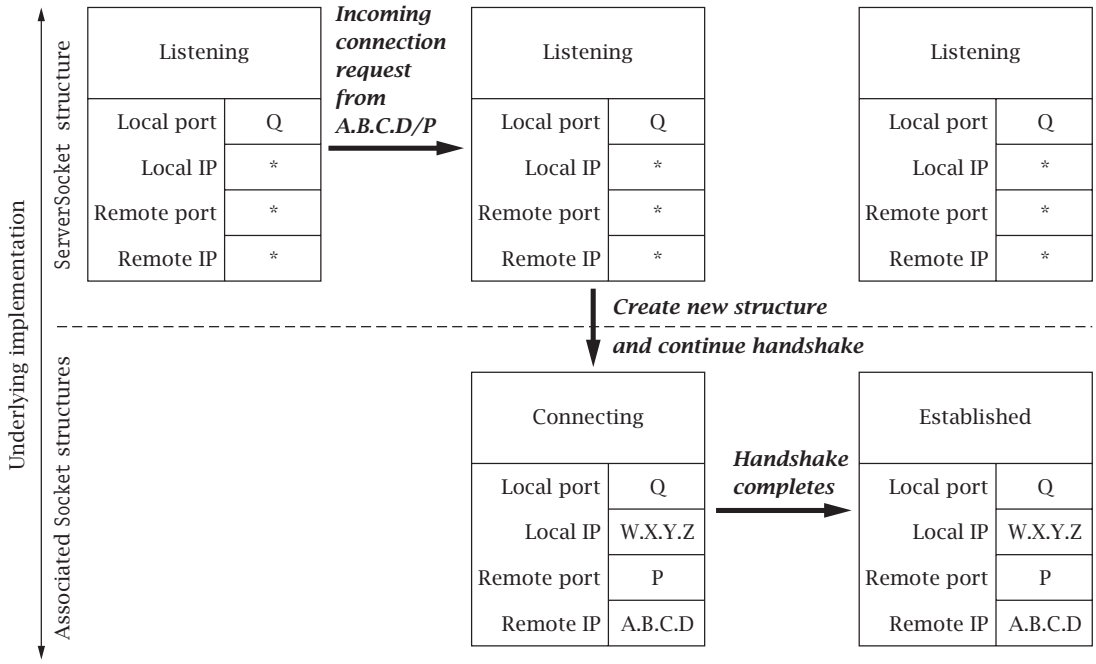


Figure 5.8: Incoming connection request processing.

When the request for a connection arrives from the client, a new socket structure is created for the connection. The new socket's addresses are filled in based on the arriving packet: the packet's destination Internet address and port (W.X.Y.Z and Q, respectively) become the local Internet address and port; the packet's source address and port (A.B.C.D and P) become the remote Internet address and port. Note that the local port number of the new socket is always the same as that of the `ServerSocket`. The new socket's state is set to "Connecting", and it is added to a list of not-quite-connected sockets associated with the socket structure of the `ServerSocket`. Note that the `ServerSocket` itself does not change state, nor does any of its address information change.

In addition to creating a new underlying socket structure, the server-side TCP implementation sends an acknowledging TCP handshake message back to the client. However, the server TCP does not consider the handshake complete until the third message of the 3-way handshake is received from the client. When that message eventually arrives, the new structure's state is set to "Established", and it is then (and only then) moved to a list of socket structures associated with the `ServerSocket` structure, which represent established connections ready to be `accept()`ed via the `ServerSocket`. (If the third handshake message fails to arrive, eventually the "Connecting" structure is deleted.)

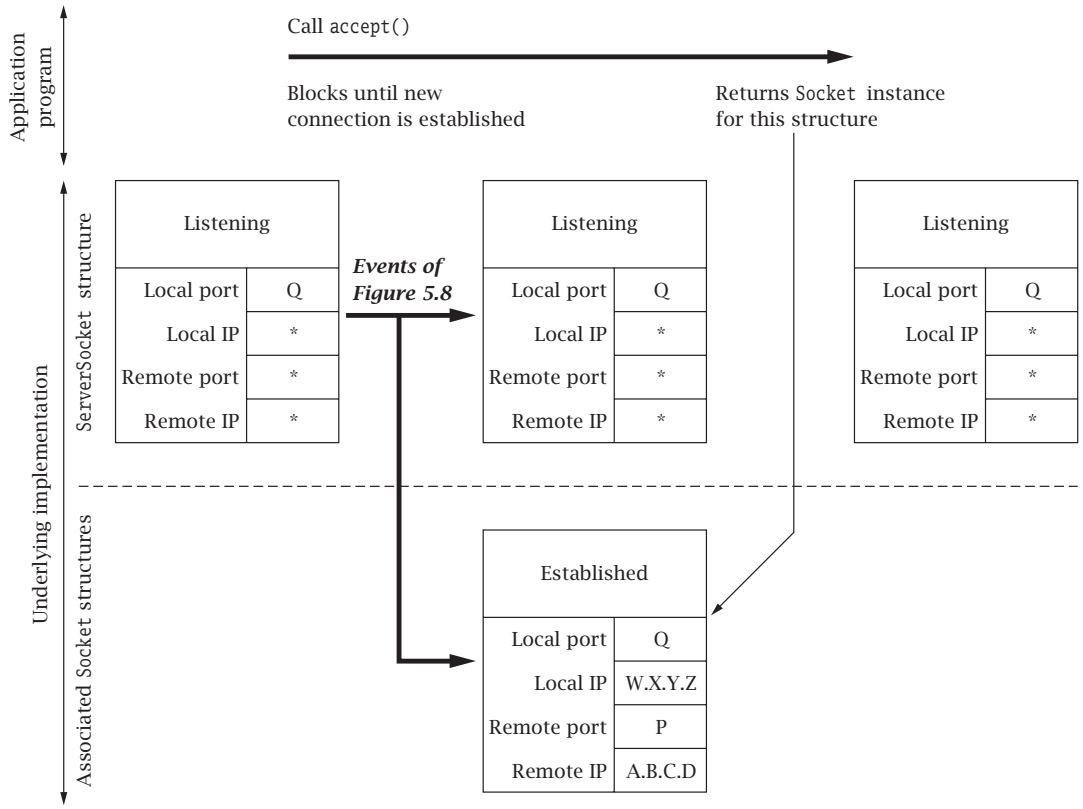


Figure 5.9: `accept()` processing.

Now we can consider (in Figure 5.9) what happens when the server program calls the `ServerSocket`'s `accept()` method. The call unblocks as soon as there is something in its associated list of socket structures for new connections. (Note that this list may already be non-empty when `accept()` is called.) At that time, one of the new connection structures is removed from the list, and an instance of `Socket` is created for it and returned as the result of the `accept()`.

It is important to note that each structure in the `ServerSocket`'s associated list represents a fully established TCP connection with a client at the other end. Indeed, the client can send data as soon as it receives the second message of the opening handshake—which may be long before the server calls `accept()` to get a `Socket` instance for it.

5.4.2 Closing a TCP Connection

TCP has a *graceful close* mechanism that allows applications to terminate a connection without having to worry about loss of data that might still be in transit. The mechanism is also designed to allow data transfers in each direction to be terminated independently, as in the

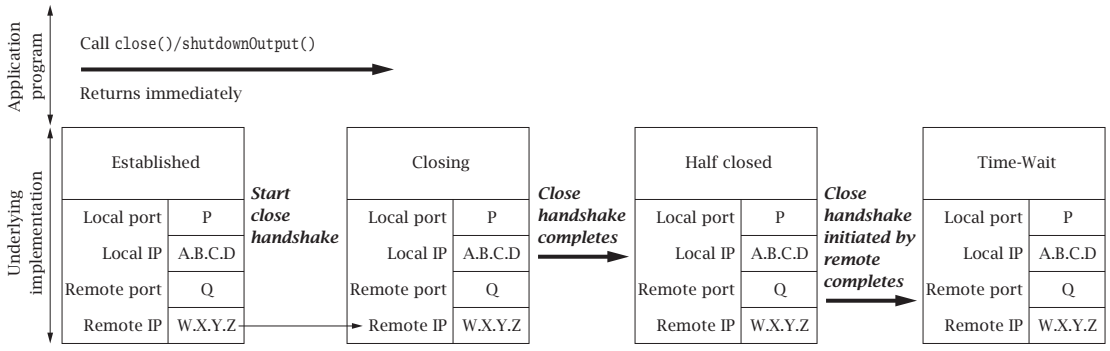


Figure 5.10: Closing a TCP connection first.

compression example of Section 4.5. It works like this: the application indicates that it is finished sending data on a connected socket by calling `close()` or by calling `shutdownOutput()`. At that point, the underlying TCP implementation first transmits any data remaining in *SendQ* (subject to available space in *RecvQ* at the other end), and then sends a closing TCP handshake message to the other end. This closing handshake message can be thought of as an end-of-transmission marker: it tells the receiving TCP that no more bytes will be placed in *RecvQ*. (Note that the closing handshake message itself is *not* passed to the receiving application, but that its position in the byte stream is indicated by `read()` returning −1.) The closing TCP waits for an acknowledgment of its closing handshake message, which indicates that all data sent on the connection made it safely to *RecvQ*. Once that acknowledgment is received, the connection is "Half closed." It is not *completely* closed until a symmetric handshake happens in the other direction—that is, until *both* ends have indicated that they have no more data to send.

The closing event sequence in TCP can happen in two ways: either one application calls `close()` (or `shutdownOutput()`) and completes its closing handshake before the other calls `close()`, or both call `close()` simultaneously, so that their closing handshake messages cross in the network. Figure 5.10 shows the sequence of events in the implementation when the application invokes `close()` *before* the other end closes. The closing handshake message is sent, the state of the socket structure is set to "Closing", and the call returns. After this point, further reads and writes on the `Socket` are disallowed (they throw an exception). When the acknowledgment for the close handshake is received, the state changes to "Half closed", where it remains until the other end's close handshake message is received. Note that if the remote endpoint goes away while the connection is in this state, the local underlying structure will stay around indefinitely. When the other end's close handshake message arrives, an acknowledgment is sent and the state is changed to "Time-Wait". Although the corresponding `Socket` instance in the application program may have long since vanished, the associated underlying structure continues to exist in the implementation for a minute or more; the reasons for this are discussed on page 107.

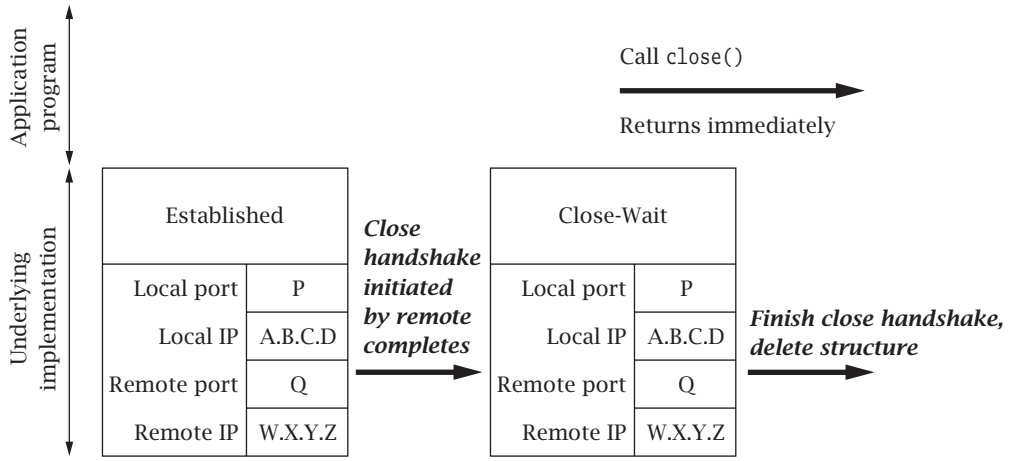


Figure 5.11: Closing after the other end closes.

Figure 5.11 shows the simpler sequence of events at the endpoint that does not close first. When the closing handshake message arrives, an acknowledgment is sent immediately, and the connection state becomes "Close-Wait." At this point, we are just waiting for the application to invoke the `Socket`'s `close()` method. When it does, the final close handshake is initiated and the underlying socket structure is deallocated, although references to its original `Socket` instance may persist in the Java program.

In view of the fact that both `close()` and `shutdownOutput()` return without waiting for the closing handshake to complete, you may wonder how the sender can be assured that sent data has actually made it to the receiving program (i.e., to *Delivered*). In fact, it is possible for an application to call `close()` or `shutdownOutput()` and have it complete successfully (i.e., not throw an Exception) *while there is still data in SendQ*. If either end of the connection then crashes before the data makes it to *RecvQ*, data may be lost without the sending application knowing about it.

The best solution is to design the application protocol so that the side that calls `close()` first does so *only after* receiving application-level assurance that its data was received. For example, when our `TCPEchoClient` program receives the echoed copy of the data it sent, there should be nothing more in transit in either direction, so it is safe to close the connection.

Java does provide a way to modify the behavior of the `Socket`'s `close()` method, namely, the `setSoLinger()` method. `setSoLinger()` controls whether `close()` waits for the closing handshake to complete before returning. It takes two parameters, a boolean that indicates whether to wait, and an integer specifying the number of seconds to wait before giving up. That is, when a timeout is specified via `setSoLinger()`, `close()` blocks until the closing handshake is completed, or until the specified amount of time passes. At the time of this writing, however, `close()` provides no indication that the closing handshake failed to complete, even if the time limit set by `setSoLinger()` expires before the closing sequence completes. In other words, `setSoLinger()` does not provide any additional assurance to the application in current implementations.

The final subtlety of closing a TCP connection revolves around the need for the Time-Wait state. The TCP specification requires that when a connection terminates, at least one of the sockets persists in the Time-Wait state for a period of time after both closing handshakes complete. This requirement is motivated by the possibility of messages being delayed in the network. If both ends' underlying structures go away as soon as both closing handshakes complete, and a *new* connection is immediately established between the same pair of socket addresses, a message from the previous connection, which happened to be delayed in the network, could arrive just after the new connection is established. Because it would contain the same source and destination addresses, the old message could be mistaken for a message belonging to the new connection, and its data might (incorrectly) be delivered to the application.

Unlikely though this scenario may be, TCP employs multiple mechanisms to prevent it, including the Time-Wait state. The Time-Wait state ensures that every TCP connection ends with a quiet time, during which no data is sent. The quiet time is supposed to be equal to twice the maximum amount of time a packet can remain in the network. Thus, by the time a connection goes away completely (i.e., the socket structure leaves the Time-Wait state and is deallocated) and clears the way for a new connection between the same pair of addresses, no messages from the old instance can still be in the network. In practice, the length of the quiet time is implementation dependent, because there is no real mechanism that limits how long a packet can be delayed by the network. Values in use range from 4 minutes down to 30 seconds or even shorter.

The most important consequence of Time-Wait is that as long as the underlying socket structure exists, no other socket is permitted to be associated with the same local port. In particular, any attempt to create a `Socket` instance using that port will throw an `IOException`.

5.5 Demultiplexing Demystified

The fact that different sockets on the same machine can have the same local address and port number is implicit in the discussions above. For example, on a machine with only one IP address, every new `Socket` instance `accept()`ed via a `ServerSocket` will have the same local port number as the `ServerSocket`. Clearly the process of deciding to which socket an incoming packet should be delivered—that is, the *demultiplexing* process—involves looking at more than just the packet's destination address and port. Otherwise there could be ambiguity about which socket an incoming packet is intended for. The process of matching an incoming packet to a socket is actually the same for both TCP and UDP, and can be summarized by the following points:

- The local port in the socket structure *must* match the destination port number in the incoming packet.
- Any address fields in the socket structure that contain the wildcard value (*) are considered to match *any* value in the corresponding field in the packet.
- If there is more than one socket structure that matches an incoming packet for all four address fields, the one that matches using the fewest wildcards gets the packet.

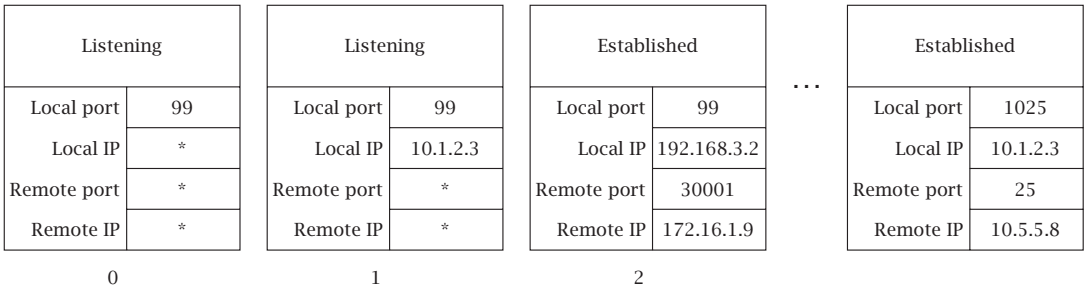


Figure 5.12: Demultiplexing with multiple matching sockets.

For example, consider a host with two IP addresses, 10.1.2.3 and 192.168.3.2, and with a subset of its active TCP socket structures, as shown in Figure 5.12. The structure labeled 0 is associated with a `ServerSocket` and has port 99 with a wildcard local address. Socket structure 1 is also for a `ServerSocket` on the same port, but with the local IP address 10.1.2.3 specified (so it will only accept connection requests to that address). Structure 2 is for a connection that was accepted via the `ServerSocket` for structure 0, and thus has the same local port number, but also has its local and remote Internet addresses filled in. Other sockets belong to other active connections. Now consider a packet with source IP address 172.16.1.10, source port 56789, destination IP address 10.1.2.3, and destination port 99. It will be delivered to the socket associated with structure 1, because that one matches with the fewest wildcards.

When a program attempts to create a socket with a particular local port number, the existing sockets are checked to make sure that no socket is already using that local port. A `Socket()` constructor will throw an exception if *any* socket matches the local port and local IP address (if any) specified in the constructor. This can cause problems in the following scenario:

1. A client program creates a `Socket` with a specific local port number, say, *P*, and uses it to communicate with a server.
2. The client closes the `Socket`, and the underlying structure goes into the Time-Wait state.
3. The client program terminates and is immediately restarted.

If the new incarnation of the client attempts to use the same local port number, the `Socket` constructor will throw an `IOException`, because of the other structure in the Time-Wait state. As of this writing, the only way around this is to wait until the underlying structure leaves the Time-Wait state.

So what determines the local/foreign address/port? For a `ServerSocket`, all constructors require the local port. The local address may be specified to the constructor; otherwise, the local address is the wildcard (*) address. The foreign address and port for a `ServerSocket` are always wildcards. For a `Socket`, all constructors require specification of the foreign address and port. The local address and/or port may be specified to the constructor. Otherwise, the local address is the address of the network interface through which the connection to the server is established, and the local port is a randomly selected, unused port number greater

than 1023. For a `Socket` instance returned by `accept()`, the local address is the destination address from the initial handshake message from the client, the local port is the local port of the `ServerSocket`, and the foreign address/port is the local address/port of the client. For a `DatagramSocket`, the local address and/or port may be specified to the constructor. Otherwise the local address is the wildcard address, and the local port is a randomly selected, unused port number greater than 1023. The foreign address and port are initially both wildcards, and remain that way unless the `connect()` method is invoked to specify particular values.

5.6 Exercises

1. The TCP protocol is designed so that simultaneous connection attempts will succeed. That is, if an application using port P and Internet address W.X.Y.Z attempts to connect to address A.B.C.D, port Q, at the same time as an application using the same address and port tries to connect to W.X.Y.Z, port P, they will end up connected to each other. Can this be made to happen when the programs use the sockets API?
2. The first example of "buffer deadlock" in this chapter involves the programs on both ends of a connection trying to send large messages. However, this is not necessary for deadlock. How could the `TCPEchoClient` from Chapter 2 be made to deadlock when it connects to the `TCPEchoServer` from that chapter?

This Page Intentionally Left Blank

Bibliography

[1] Case, J. D., Fedor, M., and Schoffstall, M. L., "Simple Network Management Protocol (SNMP)." Internet Request for Comments 1157, May 1990.

[2] Comer, Douglas E., *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture* (third edition). Prentice Hall, 1995.

[3] Comer, Douglas E., and Stevens, David L., *Internetworking with TCP/IP, Volume II: Design, Implementation, and Internals* (third edition). Prentice Hall, 1999.

[4] Comer, Douglas E., and Stevens, David L., *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications* (BSD version, second edition). Prentice Hall, 1996.

[5] Deering, S., and Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification." Internet Request for Comments 2460, December 1998.

[6] Gilligan, R., Thomson, S., Bound, J., and Stevens, W., "Basic Socket Interface Extensions for IPv6." Internet Request for Comments 2553, March 1999.

[7] Hughes, M., Shoffner, M., Hamner, D., and Bellus, U., *Java Network Programming* (second edition). Manning, 1999.

[8] International Organization for Standardization, *Information Processing Systems—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*. International Standard 8824, December 1987.

[9] Mockapetris, P., "Domain Names—Concepts and Facilities." Internet Request for Comments 1034, November 1987.

[10] Mockapetris, P., "Domain Names—Implementation and Specification." Internet Request for Comments 1035, November 1987.

[11] Peterson, L. L., and Davie, B. S., *Computer Networks: A Systems Approach* (second edition). Morgan Kaufmann, 2000.

[12] Postel, J., "Internet Protocol." Internet Request for Comments 791, September 1981.

[13] Postel, J., "Transmission Control Protocol." Internet Request for Comments 793, September 1981.

[14] Postel, J., "User Datagram Protocol." Internet Request for Comments 768, August 1980.

[15] Steedman, D., *Abstract Syntax Notation One (ASN.1)—The Tutorial and Reference*. Technology Appraisals, U.K., 1990.

[16] Stevens, W. R., *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[17] Stevens, W. R., *UNIX Network Programming: Networking APIs: Sockets and XTI* (second edition). Prentice Hall, 1997.

[18] Sun Microsystems Incorporated, "External Data Representation Standard." Internet Request for Comments 1014, June 1987.

[19] Sun Microsystems Incorporated, "Network File System Protocol Specification." Internet Request for Comments 1094, March 1989.

[20] Sun Microsystems Incorporated, "Network File System Protocol Version 3 Specification." Internet Request for Comments 1813, June 1995.

[21] The Unicode Consortium, *The Unicode Standard, Version 3.0*. Addison-Wesley Longman, 2000.

[22] Wright, G. R., and Stevens, W. R., *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

Index