Internetworking With TCP/IP

Vol III:

Client-Server Programming And Applications

BSD Socket Version

Second Edition

DOUGLAS E. COMER

and

DAVID L. STEVENS

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907

Contents

1 Introduction And Overview ... 7

1.1 Use Of TCP/IP ... 7

1.2 Designing Applications For A Distributed Environment ... 7

1.3 Standard And Nonstandard Application Protocols ... 7

1.4 An Example Of Standard Application Protocol Use ... 8

1.5 An Example Connection ... 8

1.6 Using TELNET To Access An Alternative Service ... 9

1.7 Application Protocols And Software Flexibility ... 10

1.8 Viewing Services From The Provider's Perspective ... 10

1.9 The Remainder Of This Text ... 11

1.10 Summary ... 11

2 The Client Server Model And Software Design ... 13

2.1 Introduction ... 13

2.2 Motivation ... 13

2.3 Terminology And Concepts ... 13

2.4 Summary ... 19

3 Concurrent Processing In Client-Server Software ... 21

3.1 Introduction ... 21

3.2 Concurrency In Networks ... 21

3.3 Concurrency In Servers ... 22

3.4 Terminology And Concepts ... 23

3.5 An Example Of Concurrent Process Creation ... 25

3.6 Executing New Code ... 29

3.7 Context Switching And Protocol Software Design ... 29

3.8 Concurrency And Asynchronous I/O ... 29

3.9 Summary ... 30

4 Program Interface To Protocols ... 33

4.1 Introduction ... 33

4.2 Loosely Specified Protocol Software Interface ... 33

4.3 Interface Functionality ... 33

4.4 Conceptual Interface Specification ... 34

4.5 System Calls ... 34

4.6 Two Basic Approaches To Network Communication 35
4.7 The Basic I/O Functions Available In UNIX 35
4.8 Using UNIX I/O With TCP/IP 36
4.9 Summary 37
5 The Socket Interface 39
5.1 Introduction 39
5.2 Berkeley Sockets 39
5.3 Specifying A Protocol Interface 39
5.4 The Socket Abstraction 40
5.4.2 System Data Structures For Sockets 41
5.5 Specifying An Endpoint Address 41
5.6 A Generic Address Structure 42
5.7 Major System Calls Used With Sockets 43
5.8 Utility Routines For Integer Conversion 45
5.9 Using Socket Calls In A Program 46
5.10 Symbolic Constants For Socket Call Parameters 46
5.11 Summary 47
6 Algorithms And Issues In Client Software Design 48
6.1 Introduction 48
6.2 Learning Algorithms Instead Of Details 48
6.3 Client Architecture 48
6.4 Identifying The Location Of A Server 48
6.5 Parsing An Address Argument 50
6.6 Looking Up A Domain Name 50
6.7 Looking Up A Well-Known Port By Name 51
6.8 Port Numbers And Network Byte Order 52
6.9 Looking Up A Protocol By Name 52
6.10 The TCP Client Algorithm 52
6.11 Allocating A Socket 53
6.12 Choosing A Local Protocol Port Number 53
6.13 A Fundamental Problem In Choosing A Local IP Address 53
6.14 Connecting A TCP Socket To A Server 54
6.15 Communicating With The Server Using TCP 54
6.16 Reading A Response From A TCP Connection 55
6.17 Closing A TCP Connection 6.17.1 The Need For Partial Close 56

6.17.2 A Partial Close Operation 56
6.18 Programming A UDP Client 56
6.19 Connected And Unconnected UDP Sockets 57
6.20 Using Connect With UDP 57
6.21 Communicating With A Server Using UDP 57
6.22 Closing A Socket That Uses UDP 58
6.23 Partial Close For UDP 58
6.24 A Warning About UDP Unreliability 58
6.25 Summary 58
7 Example Client Software 60
7.1 Introduction 60
7.2 The Importance Of Small Examples 60
7.3 Hiding Details 60
7.4 An Example Procedure Library For Client Programs 61
7.5 Implementation Of ConnectTCP 61
7.6 Implementation Of ConnectUDP 62
7.7 A Procedure That Forms Connections 62
7.8 Using The Example Library 65
7.9 The DAYTIME Service 65
7.10 Implementation Of A TCP Client For DAYTIME 66
7.11 Reading From A TCP Connection 67
7.12 The TIME Service 67
7.13 Accessing The TIME Service 68
7.14 Accurate Times And Network Delays 68
7.15 A UDP Client For The TIME Service 68
7.16 The ECHO Service 70
7.17 A TCP Client For The ECHO Service 70
7.18 A UDP Client For The ECHO Service 72
7.19 Summary 74
8 Algorithms And Issues In Server Software Design 77
8.1 Introduction 77
8.2 The Conceptual Server Algorithm 77
8.3 Concurrent Vs. Iterative Servers 77
8.4 Connection-Oriented Vs. Connectionless Access 77
8.5 Connection-Oriented Servers 78

8.6 Connectionless Servers........ 78
8.7 Failure, Reliability, And Statelessness........ 79
8.8 Optimizing Stateless Servers........ 79
8.9 Four Basic Types Of Servers........ 80
8.10 Request Processing Time........ 81
8.11 Iterative Server Algorithms........ 81
8.12 An Iterative, Connection-Oriented Server Algorithm........ 81
8.13 Binding To A Well-Known Address Using INADDR_ANY........ 82
8.14 Placing The Socket In Passive Mode........ 82
8.15 Accepting Connections And Using Them........ 82
8.16 An Iterative, Connectionless Server Algorithm........ 83
8.17 Forming A Reply Address In A Connectionless Server........ 83
8.18 Concurrent Server Algorithms........ 83
8.19 Master And Slave Processes........ 84
8.20 A Concurrent, Connectionless Server Algorithm........ 84
8.21 A Concurrent, Connection-Oriented Server Algorithm........ 84
8.22 Using Separate Programs As Slaves........ 85
8.23 Apparent Concurrency Using A Single Process........ 85
8.24 When To Use Each Server Type........ 86
8.25 A Summary of Server Types Iterative, Connectionless Server........ 87
8.26 The Important Problem Of Server Deadlock........ 87
8.27 Alternative Implementations........ 88
8.28 Summary........ 88
9 Iterative, Connectionless Servers (UDP)........ 91
9.1 Introduction........ 91
9.2 Creating A Passive Socket........ 91
9.3 Process Structure........ 94
9.4 An Example TIME Server........ 94
9.5 Summary........ 96
10 Iterative, Connection-Oriented Servers (TCP)........ 99
10.1 Introduction........ 99
10.2 Allocating A Passive TCP Socket........ 99
10.3 A Server For The DAYTIME Service........ 99
10.4 Process Structure........ 100
10.5 An Example DAYTIME Server........ 100

10.6 Closing Connections 102
10.7 Connection Termination And Server Vulnerability... 103
10.8 Summary... 103
11 Concurrent, Connection-Oriented Servers (TCP) 105
11.1 Introduction 105
11.2 Concurrent ECHO.. 105
11.3 Iterative Vs. Concurrent Implementations 105
11.4 Process Structure.. 105
11.5 An Example Concurrent ECHO Server 106
11.6 Cleaning Up Errant Processes 109
11.7 Summary... 110
12 Single-Process, Concurrent Servers (TCP) 111
12.1 Introduction 111
12.2 Data-driven Processing In A Server.. 111
12.3 Data-Driven Processing With A Single Process 111
12.4 Process Structure Of A Single-Process Server... 112
12.5 An Example Single-Process ECHO Server 112
12.6 Summary... 115
12 Single-Process, Concurrent Servers (TCP) 117
12.1 Introduction 117
12.2 Data-driven Processing In A Server.. 117
12.3 Data-Driven Processing With A Single Process 117
12.4 Process Structure Of A Single-Process Server... 118
12.5 An Example Single-Process ECHO Server 118
12.6 Summary... 121

1
Introduction And Overview

1.1 Use Of TCP/IP

In 1982, the TCP/IP Internet included a few hundred computers at two dozen sites concentrated primarily in North America. In 1996, over 8,000,000 computer systems attach to the Internet in over 85 countries spread across 7 continents; its size continues to double every ten months. Many of the over 60,000 networks that comprise the Internet are located outside the US.

In addition, most large corporations have chosen TCP/IP protocols for their private corporate internets, many of which are now as large as the connected Internet was twelve years ago. TCP/IP accounts for a significant fraction of networking throughout the world. Its use is growing rapidly in Europe, India, South America, and countries on the Pacific rim.

Besides quantitative growth, the past decade has witnessed an important change in the way sites use TCP/IP. Early use focused on a few basic services like electronic mail, file transfer, and remote login. More recently, browsing information on the World Wide Web has replaced file transfer as the most popular global service; Uniform Resource Locators used with Web browsers appear on billboards and television shows. In addition, many companies are designing application protocols and building private application software. In fact, over one fifth of all traffic on the connected Internet arises from applications other than well-known services. New applications rely on TCP/IP to provide basic transport services. They add rich functionality that has enhanced the Internet environment and has enabled new groups of users to benefit from connectivity.

The variety of applications using TCP/IP is staggering: it includes hotel reservation systems, applications that monitor and control offshore oil platforms, warehouse inventory control systems, applications that permit geographically distributed machines to share file access and display graphics, applications that transfer images and manage printing presses, as well as teleconferencing and multimedia systems. In addition, new applications are emerging constantly.

As corporate internets mature, emphasis shifts from building internets to using them. As a result, more programmers need to know the fundamental principles and techniques used to design and implement distributed applications.

1.2 Designing Applications For A Distributed Environment

Programmers who build applications for a distributed computing environment follow a simple guideline: they try to make each distributed application behave as much as possible like the nondistributed version of the program. In essence, the goal of distributed computing is to provide an environment that hides the geographic location of computers and services and makes them appear to be local.

For example, a conventional database system stores information on the same machine as the application programs that access it. A distributed version of such a database system permits users to access data from computers other than the one on which the data resides. If the distributed database applications have been designed well, a user will not know whether the data being accessed is local or remote.

1.3 Standard And Nonstandard Application Protocols

The TCP/IP protocol suite includes many application protocols, and new application protocols appear daily. In fact, whenever a programmer devises a distributed program that uses TCP/IP to communicate, the programmer has invented a new application protocol. Of course, some application protocols have been documented in RFCs and adopted as part of the official TCP/IP protocol suite. We refer to such protocols as *standard application protocols.* Other protocols, invented by application programmers for private use, are referred to as *nonstandard application protocols.*

Most network managers choose to use standard application protocols whenever possible; one does not invent a new application protocol when an existing protocol suffices. For example, the TCP/IP suite contains standard application protocols

for services like *file transfer, remote login,* and *electronic mail.* Thus, a programmer would use a standard protocol for such services.

1.4 An Example Of Standard Application Protocol Use

Remote login ranks among the most popular TCP/IP applications. Although a given remote login session only generates data at the speed a human can type and only receives data at the speed a human can read, remote login is the fourth highest source of packets on the connected Internet, exceeded by Web browsing, file transfer, and network news. Many users rely on remote login as part of their working environment; they do not have a direct connection to the machines that they use for most computation.

The TCP/IP suite includes a standard application protocol for remote login known as *TELNET.* The TELNET protocol defines the format of data that an application program must send to a remote machine to log onto that system and the format of messages the remote machine sends back. It specifies how character data should be encoded for transmission and how one sends special messages to control the session or abort a remote operation.

For most users, the internal details of how the TELNET protocol encodes data are irrelevant; a user can invoke software that accesses a remote machine without knowing or caring about the implementation. In fact, using a remote service is usually as easy as using a local one. For example, computer systems that run TCP/IP protocols usually include a command that users invoke to run TELNET software. On Berkeley UNIX systems, the command is named *telnet.* To invoke it, a user types:

```
telnet machine
```

where the argument *machine* denotes the domain name of the machine to which remote login access is desired. Thus, to form a TELNET connection to machine *nic.ddn.mil* a user types:

```
telnet nic.ddn.mil
```

From the user's point of view, running *telnet* converts the user's terminal into a terminal that connects directly to the remote system. If the user is running in a windowing environment, the window in which the *telnet* command has been executed will be connected to the remote machine. Once the connection has been established, the *telnet* application sends each character the user types to the remote machine, and displays each character the remote machine emits on the user's screen.

After a user invokes *telnet* and connects to a remote system, the remote system displays a prompt that requests the user to type a login identifier and a password. The prompt a machine presents to a remote user is identical to the prompt it presents to users who login on local terminals. Thus, TELNET provides each remote user with the illusion of being on a directly-connected terminal.

1.5 An Example Connection

As an example, consider what happens when a user invokes *telnet* and connects to machine *cnri.reston.va. us:*

```
telnet cnri.reston.va.us
Trying...
Connected to xnri.reston.va.us.
Escape character is '^]'.

SunOS UNIX (CNRI)

login:
```

The initial output message, *Trying...* appears while the *telnet* program converts the machine name to an IP address and tries to make a valid TCP connection to that address. As soon as the connection has been established, *telnet* prints the second and third lines, telling the user that the connection attempt has succeeded and identifying a special character that the user can type to escape from the *telnet* application temporarily if needed (e.g., if a failure occurs and the user needs to abort the connection). The notation ^] means that the user must hold the *CONTROL* key while striking the right bracket key.

The last few lines of output come from the remote machine. They identify the operating system as *SunOS,* and provide a standard login prompt. The cursor stops after the *login:* message, waiting for the user to type a valid login identifier. The user must have an account on the remote machine for the TELNET session to continue. After the user types a valid login identifier, the remote machine prompts for a password, and only permits access if the login identifier and password are valid.

1.6 Using TELNET To Access An Alternative Service

TCP/IP uses protocol port numbers to identify application services on a given machine. Software that implements a given service waits for requests at a predetermined (well-known) protocol port. For example, the remote login service accessed with the TELNET application protocol has been assigned port number 23. Thus, when a user invokes the *telnet* program, the program connects to port 23 on the specified machine.

Interestingly, the TELNET protocol can be used to access services other than the standard remote login service. To do so, a user must specify the protocol port number of the desired service. The Berkeley UNIX *telnet* command uses an optional second argument to allow the user to specify an alternative protocol port. If the user does not supply a second argument, *telnet* uses port 23. However, if the user supplies a port number, *telnet* connects to that port number. For example, if a user types:

```
telnet cnri.reston.va.us 185
```

the *telnet* program will form a connection to protocol port number 185 at machine *cnri.reston.va.* us. The machine is owned by the *Corporation For National Research Initiatives (CNRI).*

Port 185 on the machine at *CNRI* does not supply remote login service. Instead, it prints information about a recent change in the service offered, and then closes the connection.

```
telnet cnri.reston.va.us 185
Trying...
Connected to cnri.reston.va.us.
Escape character is '^]'.
******NOTICE******
The KIS client program has been moved from this machine
to info.cnri.reston.va.us (132.151.1.15) on port 185.
******************
```

Contacting port 185 on machine *info. cnri. reston. va. us* allows one to access the Knowbot Information Service. After a connection succeeds, the user receives information about the service followed by a prompt for Knowbot commands:

```
Trying...
Connected to info.cnri.reston.va.us.
Escape character is '^]'.
Knowbot Information Service
KIS Client (V2.0). Copyright CNRI 1990. All Rights Reserved.

KIS searches various Internet directory services to find
someone's street address, email address and phone number.

Type 'man' at the prompt for a complete reference with
examples. Type 'help' for a quick reference to commands.
Type 'news' for information about recent changes.
Backspace characters are '^H' or DEL
```

```
Please enter your email address in our guest book...
(Your email address?) >
```

The first three lines are the same as in the example above because they come from the *telnet* program and not the remote service. The remaining lines differ, and clearly show that the service available on port 185 is not a remote login service. The greater-than symbol on the last line serves as the prompt for Knowbot commands.

The Knowbot service searches well-known white pages directories to help a user find information about another user. For example, suppose one wanted to know the email address for David Clark, a researcher at MIT. Typing *clark* in response to the Knowbot prompt retrieves over 675 entries that each contain the name *Clark.* Most of the entries correspond to individuals with a first or last name of *Clark,* but some correspond to individuals with *Clark* in their affiliation (e.g., *Clark College).* Searching through the retrieved information reveals only one entry for a David Clark at MIT:

```
Clark, David D. (DDCI)      ddc@LCS.MIT.EDU         (617)253-6003
```

1.7 Application Protocols And Software Flexibility

The example above shows how a single piece of software, in this instance the *telnet* program, can be used to access more than one service. The design of the TELNET protocol and its use to access the Knowbot service illustrate two important points. First, the goal of all protocol design is to find fundamental abstractions that can be reused in multiple applications. In practice, TELNET suffices for a wide variety of services because it provides a basic interactive communication facility. Conceptually, the protocol used to access a service remains separate from the service itself. Second, when architects specify application services, they use standard application protocols whenever possible. The Knowbot service described above can be accessed easily because it uses the standard TELNET protocol for communication. Furthermore, because most TCP/IP software includes an application program that users can invoke to run TELNET, no additional client software is needed to access the Knowbot service. Designers who invent new interactive applications can reuse software if they choose TELNET for their access protocol. The point can be summarized:

> *The TELNET protocol provides incredible flexibility because it only defines interactive communication and not the details of the service accessed. TELNET can be used as the communication mechanism for many interactive services besides remote login.*

1.8 Viewing Services From The Provider's Perspective

The examples of application services given above show how a service appears from an individual user's point of view. The user runs a program that accesses a remote service, and expects to receive a reply with little or no delay.

From the perspective of a computer that supplies a service, the situation appears quite different. Users at multiple sites may choose to access a given service at the same time. When they do, each user expects to receive a response without delay.

To provide quick responses and handle many requests, a computer system that supplies an application service must use *concurrent processing.* That is, the provider cannot keep a new user waiting while it handles requests for the previous user. Instead, the software must process more than one request at a time.

Because application programmers do not often write concurrent programs, concurrent processing can seem like magic. A single application program must manage multiple activities at the same time. In the case of TELNET, the program that provides remote login service must allow multiple users to login to a given machine and must manage multiple active login sessions. Communication for one login session must proceed without interference from others.

The need for concurrency complicates network software design, implementation, and maintenance. It mandates new algorithms and new programming techniques. Furthermore, because concurrency complicates debugging, programmers must be especially careful to document their designs and to follow good programming practices. Finally, programmers must choose a

level of concurrency and consider whether their software will exhibit higher throughput if they increase or decrease the level of concurrency.

This text helps application programmers understand the design, construction, and optimization of network application software that uses concurrent processing. It describes the fundamental algorithms for both sequential and concurrent implementations of application protocols and provides an example of each. It considers the tradeoffs and advantages of each design. Later chapters discuss the subtleties of concurrency management and review techniques that permit a programmer to optimize throughput automatically. To summarize:

> Providing concurrent access to application services is important and difficult; many chapters of this text explain and discuss concurrent implementations of application protocol software.

1.9 The Remainder Of This Text

This text describes how to design and build distributed applications. Although it uses TCP/IP transport protocols to provide concrete examples, the discussion focuses on principles, algorithms, and general purpose techniques that apply to most network protocols. Early chapters introduce the client-server model and socket interface. Later chapters present specific algorithms and implementation techniques used in client and server software as well as interesting combinations of algorithms and techniques for managing concurrency.

In addition to its description of algorithms for client and server software, the text presents general techniques like tunneling, application-level gateways, and remote procedure calls. Finally, it examines a few standard application protocols like NFS and TELNET.

Most chapters contain example software that helps illustrate the principles discussed. The software should be considered part of the text. It shows clearly how all the details fit together and how the concepts appear in working programs.

1.10 Summary

Many programmers are building distributed applications that use TCP/IP as a transport mechanism. Before programmers can design and implement a distributed application, they need to understand the client-server model of computing, the operating system interface an application program uses to access protocol software, the fundamental algorithms used to implement client and server software, and alternatives to standard clientserver interaction including the use of application gateways.

Most network services permit multiple users to access the service simultaneously. The technique of concurrent processing makes it possible to build an application program that can handle multiple requests at the same time. Much of this text focuses on techniques for the concurrent implementation of application protocols and on the problem of managing concurrency.

FOR FURTHER STUDY

The manuals that vendors supply with their operating systems contain information on how to invoke commands that access services like *TELNET.* Many sites augment the set of standard commands with locally-defined commands. Check with your site administrator to find out about loc ally- available commands.

EXERCISES

1.1 Use TELNET from your local machine to login to another machine. How much delay, if any, do you experience when the second machine connects to the same local area network? How much delay do you notice when connected to a remote machine?

1.2 Read the vendor's manual to find out whether your local version of the TELNET software permits connection to a port on the remote machine other than the. standard port used for remote login.

1.3 Determine the set of TCP/IP services available on your local computer.

1.4 Use an FTP program to retrieve a file from a remote site. If the software does not provide statistics, estimate the transfer rate for a large file. Is the rate higher or lower than you expected?

1.5 Use the *finger* command to obtain information about users at a remote site.

2
The Client Server Model And Software Design

2.1 Introduction

From the viewpoint of an application, TCP/IP, like most computer communication protocols, merely provides basic mechanisms used to transfer data. In particular, TCP/IP allows a programmer to establish communication between two application programs and to pass data back and forth. Thus, we say that TCP/IP provides *peer-to-peer* communication. The peer applications can execute on the same machine or on different machines.

Although TCP/IP specifies the details of how data passes between a pair of communicating applications, it does not dictate when or why peer applications interact, nor does it specify how programmers should organize such application programs in a distributed environment. In practice, one organizational method dominates the use of TCP/IP to such an extent that almost all applications use it. The method is known as the *client-server* paradigm. In fact, client-server interaction has become so fundamental in peer-to-peer networking systems that it forms the basis for most computer communication.

This text uses the client-server paradigm to describe all application programming. It considers the motivations behind the client-server model, describes the functions of the client and server components, and shows how to construct both client and server software.

Before considering how to construct software, it is important to define client-server concepts and terminology. The next sections define terminology that is used throughout the text.

2.2 Motivation

The fundamental motivation for the client-server paradigm arises from the problem of rendezvous. To understand the problem, imagine a human trying to start two programs on separate machines and have them communicate. Also remember that computers operate many orders of magnitude faster than humans. After the human initiates the first program, the program begins execution and sends a message to its peer. Within a few milliseconds, it determines that the peer does not yet exist, so it emits an error message and exits. Meanwhile, the human initiates the second program. Unfortunately, when the second program starts execution, it finds that the peer has already ceased execution. Even if the two programs retry to communicate continually, they can each execute so quickly that the probability of them sending messages to one another simultaneously is low.

The client-server model solves the rendezvous problem by asserting that in any pair of communicating applications, one side must start execution and wait (indefinitely) for the other side to contact it. The solution is important because TCP/IP does not respond to incoming communication requests on its own.

> *Because TCP/IP does not provide any mechanisms that automatically create running programs when a message arrives, a program must be waiting to accept communication before any requests arrive.*

Thus, to ensure that computers are ready to communicate, most system administrators arrange to have communication programs start automatically whenever the operating system boots. Each program runs forever, waiting for the next request to arrive for the service it offers.

2.3 Terminology And Concepts

The client-server paradigm divides communicating applications into two broad categories, depending on whether the application waits for communication or initiates it. This section provides a concise, comprehensive definition of the two categories, and relies on later chapters to illustrate them and explain many of the subtleties.

2.3.1 Clients And Servers

The client-server paradigm uses the direction of initiation to categorize whether a program is a client or server. In general, an application that initiates peer-to-peer communication is called a *client.* End users usually invoke client software when they use a network service. Most client software consists of conventional application programs. Each time a client application executes, it contacts a server, sends a request, and awaits a response. When the response arrives, the client continues processing. Clients are often easier to build than servers, and usually require no special system privileges to operate.

By comparison, a *server is* any program[1] that waits for incoming communication requests from a client. The server receives a client's request, performs the necessary computation, and returns the result to the client.

2.3.2 Privilege And Complexity

Because servers often need to access data, computations, or protocol ports that the operating system protects, server software usually requires special system privileges. Because a server executes with special system privilege, care must be taken to ensure that it does not inadvertently pass privileges on to the clients that use it. For example, a file server that operates as a privileged program must contain code to check whether a given file can be accessed by a given client. The server cannot rely on the usual operating system checks because its privileged status overrides them.

Servers must contain code that handles the issues of:

- *Authentication - verifying* the identity of the client
- *Authorization - determining* whether a given client is permitted to access the service the server supplies
- *Data security - guaranteeing* that data is not unintentionally revealed or compromised
- *Privacy - keeping* information about an individual from unauthorized access
- *Protection - guaranteeing* that network applications cannot abuse system resources.

As we will see in later chapters, servers that perform intense computation or handle large volumes of data operate more efficiently if they handle requests concurrently. The combination of special privileges and concurrent operation usually makes servers more difficult to design and implement than clients. Later chapters provide many examples that illustrate the differences between clients and servers.

2.3.3 Standard Vs. Nonstandard Client Software

Chapter I describes two broad classes of client application programs: those that invoke standard TCP/IP services (e.g., electronic mail) and those that invoke services defined by the site (e.g., an institution's private database system). *Standard application services* consist of those services defined by TCP/IP and assigned well-known, universally recognized protocol port identifiers; we consider all others to be *locally-defined application services* or *nonstandard application services.*

The distinction between standard services and others is only important when communicating outside the local environment. Within a given environment, system administrators usually arrange to define service names in such a way that users cannot distinguish between local and standard services. Programmers who build network applications that will be used at other sites must understand the distinction, however, and must be careful to avoid depending on services that are only available locally.

Although TCP/IP defines many standard application protocols, most commercial computer vendors supply only a handful of standard application client programs with their TCP/IP software. For example, TCP/IP software usually includes a *remote terminal client* that uses the standard TELNET protocol for remote login, an *electronic mail client* that uses the standard SMTP protocol to transfer electronic mail to a remote system, a file *transfer client that* uses the standard FTP protocol to transfer files between two machines, and a *Web browser* that uses the standard HTTP protocol to access Web documents.

Of course, many organizations build customized applications that use TCP/IP to communicate. Customized, nonstandard applications range from simple to complex, and include such diverse services as image transmission and video teleconferencing,

---

[1] Technically, a *server* is a program and not a piece of hardware. However, computer users frequently (mis)apply the term to the computer responsible for running a particular server program. For example, they might say, "That computer is our file server," when they mean, "That computer runs our file server program."

voice transmission, remote real-time data collection, hotel and other on-line reservation systems, distributed database access, weather data distribution, and remote control of ocean-based drilling platforms.

2.3.4 Parameterization Of Clients

Some client software provides more generality than others. In particular, some client software allows the user to specify both the remote machine on which a server operates and the protocol port number at which the server is listening. For example, Chapter I shows how standard application client software can use the *TELNET* protocol to access services other than the conventional TELNET remote terminal service, as long as the program allows the user to specify a destination protocol port as well as a remote machine.

Conceptually, software that allows a user to specify a protocol port number has more input parameters than other software, so we use the term *fully parameterized client* to describe it. Many TELNET client implementations interpret an optional second argument as a port number. To specify only a remote machine, the user supplies the name of the remote machine:

telnet *machine-name*

Given only a machine name, the *telnet* program uses the well-known port for the TELNET service. To specify both a remote machine and a port on that machine, the user specifies both the machine name and the port number:

telnet *machine-name port*

Not all vendors provide full parameterization for their client application software. Therefore, on some systems, it may be difficult or impossible to use any port other than the official TELNET port. In fact, it may be necessary to modify the vendor's TELNET client software or to write new TELNET client software that accepts a port argument and uses that port. Of course, when building client software, full parameterization is recommended.

When designing client application software, include parameters that allow the user to fully specify the destination machine and destination protocol port number.

Full parameterization is especially useful when testing a new client or server because it allows testing to proceed independent of the existing software already in use. For example, a programmer can build a TELNET client and server pair, invoke them using nonstandard protocol ports, and proceed to test the software without disturbing standard services. Other users can continue to access the old TELNET service without interference during the testing.

2.3.5 Connectionless Vs. Connection-Oriented Servers

When programmers design client-server software, they must choose between two types of interaction: a *connectionless style* or a *connection-oriented style.* The two styles of interaction correspond directly to the two major transport protocols that the TCP/IP protocol suite supplies. If the client and server communicate using UDP, the interaction is connectionless; if they use TCP, the interaction is connection-oriented.

From the application programmer's point of view, the distinction between connectionless and connection-oriented interactions is critical because it determines the level of reliability that the underlying system provides. TCP provides all the reliability needed to communicate across an internet. It verifies that data arrives, and automatically retransmits segments that do not. It computes a checksum over the data to guarantee that it is not corrupted during transmission. It uses sequence numbers to ensure that the data arrives in order, and automatically eliminates duplicate packets. It provides flow control to ensure that the

sender does not transmit data faster than the receiver can consume it. Finally, TCP informs both the client and server if the underlying network becomes inoperable for any reason.

By contrast, clients and servers that use UDP do not have any guarantees about reliable delivery. When a client sends a request, the request may be lost, duplicated, delayed, or delivered out of order. Similarly, a response the server sends back to a client may be lost, duplicated, delayed, or delivered out of order. The client and/or server application programs must take appropriate actions to detect and correct such errors.

UDP can be deceiving because it provides *best effort delivery.* UDP does not introduce errors - it merely depends on the underlying IP internet to deliver packets. IP, in turn, depends on the underlying hardware networks and intermediate gateways. From a programmer's point of view, the consequence of using UDP is that it works well if the underlying internet works well. For example, UDP works well in a local environment because reliability errors seldom occur in a local environment. Errors usually arise only when communication spans a wide area internet.

Programmers sometimes make the mistake of choosing connectionless transport (i.e., UDP), building an application that uses it, and then testing the application software only on a local area network. Because a local area network seldom or never delays packets, drops them, or delivers them out of order, the application software appears to work well. However, if the same software is used across a wide area internet, it may fail or produce incorrect results.

Beginners, as well as most experienced professionals, prefer to use the connection oriented style of interaction. A connection-oriented protocol makes programming simpler, and relieves the programmer of the responsibility to detect and correct errors. In fact, adding reliability to a connectionless internet message protocol like UDP is a nontrivial undertaking that usually requires considerable experience with protocol design.

Usually, application programs only use UDP if: (1) the application protocol specifies that UDP must be used (presumably, the application protocol has been designed to handle reliability and delivery errors), (2) the application protocol relies on hardware broadcast or multicast, or (3) the application cannot tolerate the computational overhead or delay required for TCP virtual circuits. We can summarize:

> *When designing client-server applications, beginners are strongly advised to use TCP because it provides reliable, connection-oriented communication. Programs only use UDP if the application protocol handles reliability, the application requires hardware broadcast or multicast, or the application cannot tolerate virtual circuit overhead.*

2.3.6 Stateless Vs. Stateful Servers

Information that a server maintains about the status of ongoing interactions with clients is called *state information.* Servers that do not keep any state information are called *stateless servers;* others are called *stateful servers.*

The desire for efficiency motivates designers to keep state information in servers. Keeping a small amount of information in a server can reduce the size of messages that the client and server exchange, and can allow the server to respond to requests quickly. Essentially, state information allows a server to remember what the client requested previously and to compute an incremental response as each new request arrives. By contrast, the motivation for statelessness lies in protocol reliability: state information in a server can become incorrect if messages are lost, duplicated, or delivered out of order, or if the client computer crashes and reboots. If the server uses incorrect state information when computing a response, it may respond incorrectly.

2.3.7 A Stateful File Server Example

An example will help explain the distinction between stateless and stateful servers. Consider a file server that allows clients to remotely access information kept in the files on a local disk. The server operates as an application program. It waits for a client to contact it over the network. The client sends one of two request types. It either sends a request to extract data from a specified file or a request to store data in a specified file. The server performs the requested operation and replies to the client.

On one hand, if the file server is stateless, it maintains no information about the transactions. Each message from a client that requests the server to extract data from a file must specify the complete file name (the name could be quite lengthy), a position in the file from which the data should be extracted, and the number of bytes to extract. Similarly, each message that requests the server to store data in a file must specify the complete file name, a position in the file at which the data should be stored, and the data to store.

On the other hand, if the file server maintains state information for its clients, it can eliminate the need to pass file names in each message. The server maintains a table that holds state information about the file currently being accessed. Figure 2.1 shows one possible arrangement of the state information.

Handle	File Name	Current Position
1	test.program.c	0
2	tcp.book.doc	456
3	dept.budget.text	38
4	tetris.exe	128

Figure 2.1 Example table of state information for a stateful file server. To keep messages short, the server assigns a handle to each file. The handle appears in messages instead of a file name.

When a client first opens a file, the server adds an entry to its state table that contains the name of the file, a *handle* (a small integer used to identify the file), and a current position in the file (initially zero). The server then sends the handle back to the client for use in subsequent requests. Whenever the client wants to extract additional data from the file, it sends a small message that includes the handle. The server uses the handle to look up the file name and current file position in its state table. The server increments the file position in the state table, so the next request from the client will extract new data. Thus, the client can send repeated requests to move through the entire file. When the client finishes using a file, it sends a message informing the server that the file will no longer be needed. In response, the server removes the stored state information. As long as all messages travel reliably between the client and server, a stateful design makes the interaction more efficient. The point is:

In an ideal world, where networks deliver all messages reliably and computers never crash, having a server maintain a small amount of state information for each ongoing interaction can make messages smaller and processing simpler.

Although state information can improve efficiency, it can also be difficult or impossible to maintain correctly if the underlying network duplicates, delays, or delivers messages out of order (e.g., if the client and server use UDP to communicate). Consider what happens to our file server example if the network duplicates a *read* request. Recall that the server maintains a notion of file position in its state information. Assume that the server updates its notion of file position each time a client extracts data from a file. If the network duplicates a *read* request, the server will receive two copies. When the first copy arrives, the server extracts data from the file, updates the file position in its state information, and returns the result to the client. When the second copy arrives, the server extracts additional data, updates the file position again, and returns the new data to the client. The client may view the second response as a duplicate and discard it, or it may report an error because it received two different responses to a single request. In either case, the state information at the server can become incorrect because it disagrees with the client's notion of the true state.

When computers reboot, state information can also become incorrect. If a client crashes after performing an operation that creates additional state information, the server may never receive messages that allow it to discard the information. Eventually, the accumulated state information exhausts the server's memory. In our file server example, if a client opens 100 files and then crashes, the server will maintain 100 useless entries in its state table forever.

A stateful server may also become confused (or respond incorrectly) if a new client begins operation after a reboot using the same protocol port numbers as the previous client that was operating when the system crashed. It may seem that this problem can be overcome easily by having the server erase previous information from a client whenever a new request for interaction

arrives. Remember, however, that the underlying internet may duplicate and delay messages, so any solution to the problem of new clients reusing protocol ports after a reboot must also handle the case where a client starts normally, but its first message to a server becomes duplicated and one copy is delayed.

In general, the problems of maintaining correct state can only be solved with complex protocols that accommodate the problems of unreliable delivery and computer system restart. To summarize:

> *In a real internet, where machines crash and reboot, and messages can be lost, delayed, duplicated, or delivered out of order, stateful designs lead to complex application protocols that are difficult to design, understand, and program correctly.*

2.3.8 Statelessness Is A Protocol Issue

Although we have discussed statelessness in the context of servers, the question of whether a server is stateless or stateful centers on the application protocol more than the implementation. If the application protocol specifies that the meaning of a particular message depends in some way on previous messages, it may be impossible to provide a stateless interaction.

In essence, the issue of statelessness focuses on whether the application protocol assumes the responsibility for reliable delivery. To avoid problems and make the interaction reliable, an application protocol designer must ensure that each message is completely unambiguous. That is, a message cannot depend on being delivered in order, nor can it depend on previous messages having been delivered. In essence, the protocol designer must build the interaction so the server gives the same response no matter when or how many times a request arrives. Mathematicians use the term *idempotent* to refer to a mathematical operation that always produces the same result. We use the term to refer to protocols that arrange for a server to give the same response to a given message no matter how many times it arrives.

> *In an internet where the underlying network can duplicate, delay or deliver messages out of order or where computers running client applications can crash unexpectedly, the server should be stateless. The server can only be stateless if the application protocol is designed to make operations idempotent.*

2.3.9 Servers As Clients

Programs do not always fit exactly into **the definition of client or server.** A server program may need to access network services that require it to act as a client. For example, suppose our file server program needs to obtain the time of day so it can stamp files with the time of access. Also suppose that the system on which it operates does not have a time-of-day clock. To obtain the time, the server acts as a client by sending a request to a time-of-day server as Figure 2.2 shows.

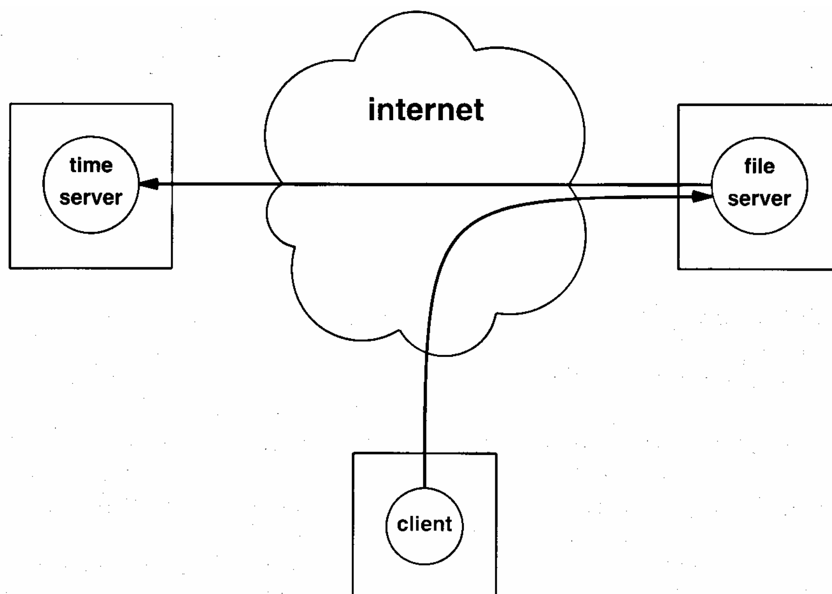

Figure 2.2 A file server program acting as a client to a time server. When the time server replies, the file server will finish its computation and return the result to the original client.

In a network environment that has many available servers, it is not unusual to find a server for one application acting as a client for another. Of course, designers must be careful to avoid circular dependencies among servers.

2.4 Summary

The client-server paradigm classifies a communicating application program as either a client or a server depending on whether it initiates communication. In addition to client and server software for standard applications, many TCP/IP users build client and server software for nonstandard applications that they define locally.

Beginners and most experienced programmers use TCP to transport messages between the client and server because it provides the reliability needed in an internet environment. Programmers only resort to UDP if TCP cannot solve the problem.

Keeping state information in the server can improve efficiency. However, if clients crash unexpectedly or the underlying transport system allows duplication, delay, or packet loss, state information can consume resources or become incorrect. Thus, most application protocol designers try to minimize state information. A stateless implementation may not be possible if the application protocol fails to make operations idempotent.

Programs cannot be divided easily into client and server categories because many programs perform both functions. A program that acts as a server for one service can act as a client to access other services.

FOR FURTHER STUDY

Stevens [1990] briefly describes the client-server model and gives UNIX examples. Other examples can be found by consulting applications that accompany various vendors' operating systems.

EXERCISES

2.1 Which of your local implementations of standard application clients are fully parameterized? Why is full parameterization needed?

2.2 Are standard application protocols like TELNET, FTP, SMTP, and NFS (Network File System) connectionless or connection-oriented?

2.3 What does TCP/IP specify should happen if no server exists when a client request arrives? (Hint: look at ICMP.) What happens on your local system?

2.4 Write down the data structures and message formats needed for a stateless file server. What happens if two or more clients access the same file? What happens if a client crashes before closing a file?

2.5 Write down the data structures and message formats needed for a stateful file server. Use the operations open, read, write, and close to access files. Arrange for open to return an integer used to access the file in read and write operations. How do you distinguish duplicate open requests from a client that sends an open, crashes, reboots, and sends an open again?

2.6 In the previous exercise, what happens in your design if two or more clients access the same file? What happens if a client crashes before closing a file?

2.7 Examine the NFS remote file access protocol carefully to identify which operations are idempotent. What errors can result if messages are lost, duplicated, or delayed?

3
Concurrent Processing In Client-Server Software

3.1 Introduction

The previous chapter defines the client-server paradigm. This chapter extends the notion of client-server interaction by discussing concurrency, a concept that provides much of the power behind client-server interactions but also makes the software difficult to design and build. The notion of concurrency also pervades later chapters, which explain in detail how servers provide concurrent access.

In addition to discussing the general concept of concurrency, this chapter also reviews the facilities that an operating system supplies to support concurrent process execution. It is important to understand the functions described in this chapter because they appear in many of the server implementations in later chapters.

3.2 Concurrency In Networks

The term *concurrency* refers to real or apparent simultaneous computing. For example, a multi-user computer system can achieve concurrency by *time-sharing,* a design that arranges to switch a single processor among multiple computations quickly enough to give the appearance of simultaneous progress; or by *multiprocessing,* a design in which multiple processors perform multiple computations simultaneously.

Concurrent processing is fundamental to distributed computing and occurs in many forms. Among machines on a single network, many pairs of application programs can communicate concurrently, sharing the network that interconnects them. For example, application A on one machine may communicate with application B on another machine, while application C on a third machine communicates with application D on a fourth. Although they all share a single network, the applications appear to proceed as if they operate independently. The network hardware enforces access rules that allow each pair of communicating machines to exchange messages. The access rules prevent a given pair of applications from excluding others by consuming all the network bandwidth.

Concurrency can also occur within a given computer system. For example, multiple users on a timesharing system can each invoke a client application that communicates with an application on another machine. One user can transfer a file while another user conducts a remote login session. From a user's point of view, it appears that all client programs proceed simultaneously.

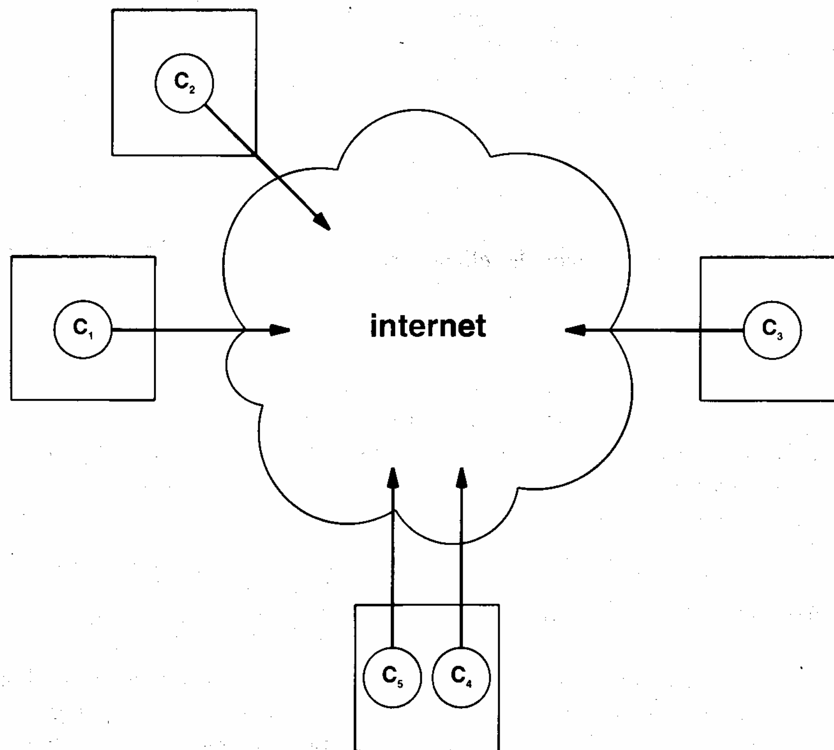

Figure 3.1 Concurrency among client programs occurs when users execute them on multiple machines simultaneously or when a multitasking operating system allows multiple copies to execute concurrently on a single computer.

In addition to concurrency among clients on a single machine, the set of all clients on a set of machines can execute concurrently. Figure 3.1 illustrates concurrency among client programs running on several machines.

Client software does not usually require any special attention or effort on the part of the programmer to make it usable concurrently. The application programmer designs and constructs each client program without regard to concurrent execution; concurrency among multiple client programs occurs automatically because the operating system allows multiple users to each invoke a client concurrently. Thus, the individual clients operate much like any conventional program. To summarize:

Most client software achieves concurrent operation because the underlying operating system allows users to execute client programs concurrently or because users on many machines each execute client software simultaneously. An individual client program operates like any conventional program; it does not manage concurrency explicitly.

3.3 Concurrency In Servers

In contrast to concurrent client software, concurrency within a server requires considerable effort. As figure 3.2 shows, a single server program must handle incoming requests concurrently.

To understand why concurrency is important, consider server operations that require substantial computation or communication. For example, think of a remote login server. It it operates with no concurrency, it can handle only one remote login at a time. Once a client contacts the server, the server must ignore or refuse subsequent requests until the first user finishes. Clearly, such a design limits the utility of the server, and prevents multiple remote users from accessing a given machine at the same time.

Chapter 8 discusses algorithms and design issues for concurrent servers, showing how they operate in principle. Chapters 9 through 13 each illustrate one of the algorithms, describing the design in more detail and showing code for a working server. The remainder of this chapter concentrates on terminology and basic concepts used throughout the text.

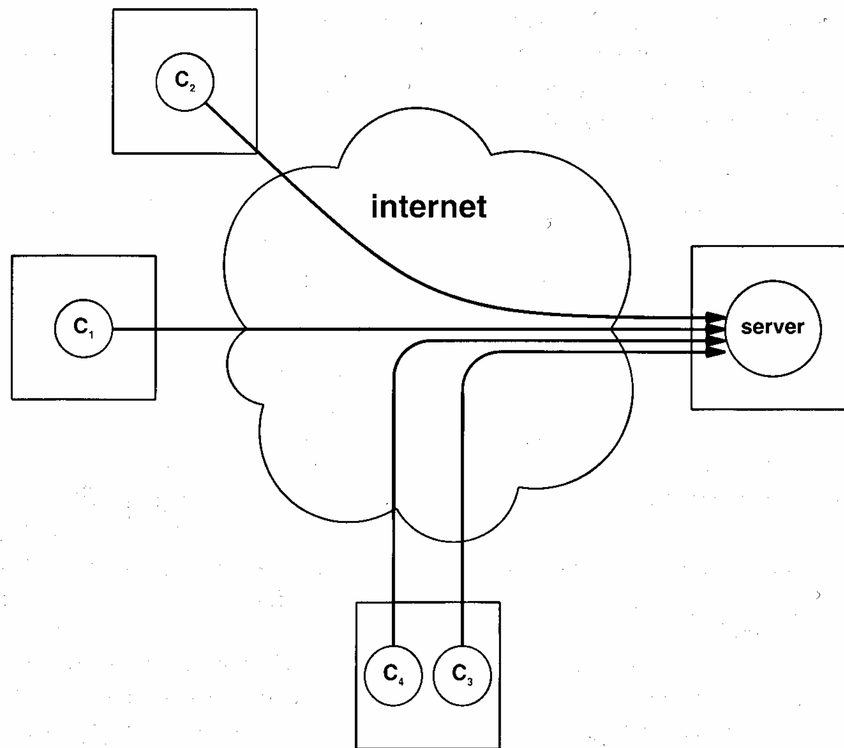

Figure 3.2 Server software must be explicitly programmed to handle concurrent requests because multiple clients contact a server using its single, well-known protocol port.

3.4 Terminology And Concepts

Because few application programmers have experience with the design of concurrent programs, understanding concurrency in servers can be challenging. This section explains the basic concept of concurrent processing and shows how an operating system supplies it. It gives examples that illustrate concurrency, and defines terminology used in later chapters.

3.4.1 The Process Concept

In concurrent processing systems, the process abstraction defines the fundamental unit of computation[1]. The most essential information associated with a process is an *instruction pointer* that specifies the address at which the process is executing. Other information associated with a process includes the identity of the user that owns it, the compiled program that it is executing, and the memory locations of the process' program text and data areas.

A process differs from a program because the process concept includes only the active execution of a computation, not the code. After the code has been loaded into a computer, the operating system allows one or more processes to execute it. In particular, a concurrent processing system allows multiple processes to execute the same piece of code "at the same time." This means that multiple processes may each be executing at some point in the code. Each process proceeds at its own rate, and each may begin or finish at an arbitrary time. Because each has a separate instruction pointer that specifies which instruction it will execute next, there is never any confusion.

---

[1] Some systems use the terms *task, job,* or *thread* instead of *process.*

Of course, on a uniprocessor architecture, the single CPU can only execute one process at any instant in time. The operating system makes the computer appear to perform more than one computation at a time by switching the CPU among all executing processes rapidly. From a human observer's point of view, many processes appear to proceed simultaneously. In fact, one process proceeds for a short time, then another process proceeds for a short time, and so on. We use the term *concurrent execution* to capture the idea. It means "apparently simultaneous execution." On a uniprocessor, the operating system handles concurrency, while on a multiprocessor, all CPUs can execute processes simultaneously.

The important concept is:

> *Application programmers build programs for a concurrent environment without knowing whether the underlying hardware consists of a uniprocessor or a multiprocessor.*

3.4.2 Programs vs. Processes

In a concurrent processing system, a conventional application program is merely a special case: it consists of a piece of code that is executed by exactly one process at a time. The notion of *process* differs from the conventional notion of *program* in other ways. For example, most application programmers think of the set of variables defined in the program as being associated with the code. However, if more than one process executes the code concurrently, it is essential that each process has its own copy of the variables. To understand why, consider the following segment of C code that prints the integers from 1 to 10:

```
for ( i=1 ; i <= 10 ; i++)
    printf("%d\n", i);
```

The iteration uses an index variable, i. In a conventional program, the programmer thinks of storage for variable i as being allocated with the code. However, if two or more processes execute the code segment concurrently, one of them may be on the sixth iteration when the other starts the first iteration. Each must have a different value for i. Thus, each process must have its own copy of variable i or confusion will result, To summarize:

> *When multiple processes execute a piece of code concurrently, each process has its own, independent copy of the variables associated with the code.*

3.4.3 Procedure Calls

In a procedure-oriented language, like Pascal or C, executed code can contain calls to subprograms (procedures or functions). Subprograms accept arguments, compute a result, and then return just after the point of the call. If multiple processes execute code concurrently, they can each be at a different point in the sequence of procedure calls. One process, A, can begin execution, call a procedure, and then call a second-level procedure before another process, B, begins. Process B may return from a first-level procedure call just as process A returns from a second-level call.

The run-time system for procedure-oriented programming languages uses a stack mechanism to handle procedure calls. The run-time system pushes a *procedure activation record* on the stack whenever it makes a procedure call. Among other things, the activation record stores information about the location in the code at which the procedure call occurs. When the procedure finishes execution, the run-time system pops the activation record from the top of the stack and returns to the procedure from which the call occurred. Analogous to the rule for variables, concurrent programming systems provide separation between procedure calls in executing processes:

When multiple processes execute a piece of code concurrently, each has its own run-time stack of procedure activation records.

3.5 An Example Of Concurrent Process Creation

3.5.1 A Sequential C Example

The following example illustrates concurrent processing in the UNIX operating system. As with most computational concepts, the programming language syntax is trivial; it occupies only a few lines of code. For example, the following code is a conventional C program that prints the integers from I to 5 along with their sum:

```
/* sum.c - A conventional C program that sum integers from 1 to 5 */

#include <stdlib.h>
#include <stdio.h>
int sum;                                    /* sum is a global variable */
main () {
         int i;                             /* i is a local variable    */
         sum = 0;
         for (i=1 ; i <=5 ; i++) {          /* iterate i from 1 to 5    */
                 printf("The value of i is %d\n", i);
                 fflush(stdout);            /* flush the buffer         */
         sum += i;
        }
        printf ("The sum is %d\n", sum);
        exit(0);                            /* terminate the program */
}
```

When executed, the program emits six lines of output:

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
```

3.5.2 A Concurrent Version

To create a new process in UNIX, a program calls the system function *fork*[2]. In essence, fork divides the running program into two (almost) identical processes, both executing at the same place in the same code. The two processes continue just as if two users had simultaneously started two copies of the application. For example, the following modified version of the above example calls fork to create a new process. (Note that although the introduction of concurrency changes the meaning of the program completely, the call to fork occupies only a single line of code.)

```
#include <stdlib.h>
#include <stdio.h>
int  sum;

main() {
        int i;
        sun = 0;
        fork();                      /* create a new process */
        for (i=1 ; i<=5 ; i++) {
                printf ("The value of i is %d\n", i);
                fflush(stdout);
                sum += i;
        }
        printf ("The sum is %d\n", sum);
        exit (0)
}
```

When a user executes the concurrent version of the program, the system begins with a single process executing the code. However, when the process reaches the call to *fork,* the system duplicates the process and allows both the original process and the newly created process to execute. Of course, each process has its own copy of the variables that the program uses. In fact, the easiest way to envision what happens is to imagine that the system makes a second copy of the entire running program. Then imagine that both copies run Oust as if two users had both simultaneously executed the program). To summarize:

> *To understand the fork function, imagine that fork causes the operating system to make a copy of the executing program and allows both copies to run at the same time.*

On one particular uniprocessor system, the execution of our example concurrent program produces twelve lines of output:

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
```

[2] To a programmer, the call to *fork* looks and acts like an ordinary function call in C. It is written `fork()`. At run-time, however, control passes to the operating system, which creates a new process.

```
The sum is 15
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
```

On the hardware being used, the first process executed so rapidly that it was able to complete execution before the second process ran at all. Once the first process completed, the operating system switched the processor to the second process, which also ran to completion. The entire run took less than a second. The operating system overhead incurred in switching between processes and handling system calls, including the call to fork and the calls required to write the output, accounted for less than 20% of the total time.

3.5.3 Timeslicing

In the example program, each process performed a trivial amount of computation as it iterated through a loop five times. Therefore, once a process gained control of the CPU, it quickly ran to completion. If we examine concurrent processes that perform substantially more computation, an interesting phenomenon occurs: the operating system allocates the available CPU power to each one for a short time before moving on to the next. We use the term *timeslicing* to describe systems that share the available CPU among several processes concurrently. For example, if a timeslicing system has only one CPU to allocate and a program divides into two processes, one of the processes will execute for a while, then the second will execute for a while, then the first will execute again, and so on. If the timeslicing system has many processes, it runs each for a short time before it runs the first one again.

A timeslicing mechanism attempts to allocate the available processing equally among all available processes. If only two processes are eligible to execute and the computer has a single processor, each receives approximately 50% of the CPU. If N processes are eligible on a computer with a single processor, each receives approximately 1/N of the CPU. Thus, all processes appear to proceed at an equal rate, no matter how many processes execute. With many processes executing, the rate is low; with few, the rate is high.

To see the effect of timeslicing, we need an example program in which each process executes longer than the allotted timeslice. Extending the concurrent program above to iterate 10,000 times instead of 5 times produces:

```
#include <stdlib.h>
#include <stdio.h>
int sum;

main() {
        int i;

        sum = 0;
        fork();
        for (i=1 ; i <=10000 ; i++) {
                printf("The value of i is %d\n", i);
                fflush(stdout);
                sum += i;
```

```
        }
        printf ("The total is %d\n", sum);
        exit (0)
}
```

When the resulting concurrent program is executed on the same system as before, it emits 20,002 lines of output. However, instead of all output from the first process followed by all output from the second process, output from both processes is mixed together. In one run, the first process iterated 74 times before the second process executed at all. Then the second process iterated 63 times before the system switched back to the first process. On subsequent timeslices, the processes each received enough CPU service to iterate between 60 and 90 times. Of course, the two processes compete with all other processes executing on the computer, so the apparent rate of execution varies slightly depending on the mix of programs running.

3.5.4 Making Processes Diverge

So far, we have said that *fork* can be used to create a new process that executes exactly the same code as the original process. Creating a truly identical copy of a running program is neither interesting nor useful because it means that both copies perform exactly the same computation. In practice, the process created by *fork* is not absolutely identical to the original process: it differs in one small detail. *Fork* is a function that returns a value to its caller. When the function call returns, the value returned to the original process differs from the value returned to the newly created process. In the newly created process, the *fork* returns zero; in the original process, *fork* returns a small positive integer that identifies the newly created process. Technically, the value returned is called a *process identifier* or *process id*[3].

Concurrent programs use the value returned by *fork* to decide how to proceed. In the most common case, the code contains a conditional statement that tests to see if the value returned is nonzero:

```
#include <stdlib.h>

int sum;
main() {
        int pid;

        sum = 0;
        pid = fork();
        if (pid != 0) {       /* original process */
           printf("The original process prints this.\n");
        } else {              /* newly created process */
           printf ("The new process prints this. \n")
        }
        exit(0);
}
```

In the example code, variable pid records the value returned by the call to fork. Remember that each process has its own copy of all variables, and that fork will either return zero (in the newly created process) or nonzero (in the original process). Following the call to fork, the if statement checks variable pid to see whether the original or the newly created process is

[3] Many programmers abbreviate process *id* as *pid*.

executing. The two processes each print an identifying message and exit. When the program runs, two messages appear: one from the original process and one from the newly created process. To summarize:

The value returned by fork differs in the original and newly created processes; concurrent programs use the difference to allow the new process to execute different code than the original process.

3.6 Executing New Code

UNIX provides a mechanism that allows any process to execute an independent, separately -compiled program. The mechanism that UNIX uses is a system call, *execve*[4], that takes three arguments: the name of a file that contains an executable object program (i.e., a program that has been compiled), a pointer to a list of string arguments to pass to the program, and a pointer to a list of strings that comprise what UNIX calls the *environment.*

Execve replaces the code that the currently executing process runs with the code from the new program. The call does not affect any other processes. Thus, to create a new process that executes the object code from a file, a process must call fork and *execve.* For example, whenever the user types a command to one of the UNIX command interpreters, the command interpreter uses fork to create a new process for the command and *execve* to execute the code.

Execve is especially important for servers that handle diverse services. To keep the code for each service separate from the code for other services, a programmer can build, write, and compile each service as a separate program. When the server needs to handle a particular service, it can use fork and *execve* to create a process that runs one of the programs. Later chapters discuss the idea in more detail, and show examples of how servers use *execve.*

3.7 Context Switching And Protocol Software Design

Although the concurrent processing facilities that operating systems provide make programs more powerful and easier to understand, they do have computational cost. To make sure that all processes proceed concurrently, the operating system uses timeslicing, switching the CPU (or CPUs) among processes so fast that it appears to a human that the processes execute simultaneously.

When the operating system temporarily stops executing one process and switches to another, a *context switch* has occurred. Switching process context requires use of the CPU, and while the CPU is busy switching, none of the application processes receives any service. Thus, we view context switching as overhead needed to support concurrent processing.

To avoid unnecessary overhead, protocol software should be designed to minimize context switching. In particular, programmers must always be careful to ensure that the benefits of introducing concurrency into a server outweigh the cost of switching context among the concurrent processes. Later chapters discuss the use of concurrency in server software, present nonconcurrent designs as well as concurrent ones, and describe circumstances that justify the use of each.

3.8 Concurrency And Asynchronous I/O

In addition to providing support for concurrent use of the CPU, some operating systems allow a single application program to initiate and control concurrent input and output operations. In BSD UNIX, the *select* system call provides a fundamental operation around which programmers can build programs that manage concurrent I/O. In principle, *select* is easy to understand: it allows a program to ask the operating system which I/O devices are ready for use.

As an example, imagine an application program that reads characters from a TCP connection and writes them to the display screen. The program might also allow the user to type commands on the keyboard to control how the data is displayed. Because a user seldom (or never) types commands, the program cannot wait for input from the keyboard - it must continue to read and display text from the TCP connection. However, if the program attempts to read from the TCP connection and no data is

---

[4] Some versions of UNIX use the older name, exec.

available, the program will block. The user may type a command while the program is blocked waiting for input on the TCP connection. The problem is that the application cannot know whether input will arrive from the keyboard or the TCP connection first. To solve the dilemma, a UNIX program calls *select.* In doing so, it asks the operating system to let it know which source of input becomes available first. The call returns as soon as a source is ready, and the program reads from that source. For now, it is only important to understand the idea behind *select;* later chapters present the details and illustrate its use.

3.9 Summary

Concurrency is fundamental to TCP/IP applications because it allows users to access services without waiting for one another. Concurrency in clients arises easily because multiple users can execute client application software at the same time. Concurrency in servers is much more difficult to achieve because server software must be programmed explicitly to handle requests concurrently.

In UNIX, a program creates an additional process using the *fork* system call. We imagine that the call to *fork* causes the operating system to duplicate the program, causing two copies to execute instead of one. Technically, *fork* is a function call because it returns a value. The only difference between the original process and a process created by *fork* lies in the value that the call returns. In the newly created process, the call returns zero; in the original process, it returns the small, positive integer process id of the newly created process. Concurrent programs use the returned value to make new processes execute a different part of the program than the original process. A process can call *execve* at any time to have the process execute code from a separately-compiled program.

Concurrency is not free. When an operating system switches context from one process to another, the system uses the CPU. Programmers who introduce concurrency into server designs must be sure that the benefits of a concurrent design outweigh the additional overhead introduced by context switching.

The *select* call permits a single process to manage concurrent I/O. A process uses *select* to find out which I/O device becomes ready first.

FOR FURTHER STUDY

Many texts on operating systems describe concurrent processing. Peterson and Silberschatz [1985] covers the general topic. Comer [19841 discusses the implementation of processes, message passing, and process coordination mechanisms. Leffler *et. al.* [1989] describes 4.3 BSD UNIX.

EXERCISES

3.1 Run the example programs on your local computer system. Approximately how many iterations of the output loop can a process make in a single timeslice?

3.2 Write a concurrent program that starts five processes. Arrange for each process to print a few lines of output and then halt.

3.3 Find out how systems other than UNIX create concurrent processes.

3.4 Read more about the UNIX *fork* function. What information does the newly created process share with the original process?

3.5 Write a program that uses *execve* to change the code a process executes.

3.6 Write a program that uses *select* to read from two terminals (serial lines), and displays the results on a screen with labels that identify the source.

3.7 Rewrite the program in the previous exercise so it does not use *select*. Which version is easier to understand? more efficient? easier to terminate cleanly?

4
Program Interface To Protocols

4.1 Introduction

Previous chapters describe the client-server model of interaction for communicating programs and discuss the relationship between concurrency and communication. This chapter considers general properties of the interface an application program uses to communicate in the client-server model. The following chapter illustrates these properties by giving details of a specific interface.

4.2 Loosely Specified Protocol Software Interface

In most implementations, TCP/IP protocol software resides in the computer's operating system. Thus, whenever an application program uses TCP/IP to communicate, it must interact with the operating system to request service. From a programmer's point of view, the routines the operating system supplies define the interface between the application and the protocol software, the *application interface.*

TCP/IP was designed to operate in a multi-vendor environment. To remain compatible with a wide variety of machines, TCP/IP designers carefully avoided choosing any vendor's internal data representation. In addition, the TCP/IP standards carefully avoid specifying the application interface in terms of features available only on a single vendor's operating system. Thus, the interface between TCP/IP and applications that use it has been *loosely specified.* In other words:

> *The TCP/IP standards do not specify the details of how application software interfaces with TCP/IP protocol software; they only suggest the required functionality, and allow system designers to choose the details.*

4.2.1 Advantages And Disadvantages

Using a loose specification for the protocol interface has advantages and disadvantages. On the positive side, it provides flexibility and tolerance. It allows designers to implement TCP/IP using operating systems that range from the simplest systems available on personal computers to the sophisticated systems used on supercomputers. More important, it means designers can use either a procedural or message-passing interface style (whichever style the operating system supports).

On the negative side, a loose specification means that designers can make the interface details different for each operating system. As vendors add new interfaces that differ from existing interfaces, application programming becomes more difficult and applications become less portable across machines. Thus, while system designers favor a loose specification, application programmers, desire a restricted specification because it means applications can be compiled for new machines without change.

In practice, only a few TCP/IP interfaces exist. The University of California at Berkeley defined an interface for the Berkeley UNIX operating system that has become known as the *socket interface,* or *sockets.* AT&T defined an interface for System V UNIX known by the acronym TLI[1]. A few other interfaces have been defined, but none has gained wide acceptance yet.

4.3 Interface Functionality

Although TCP/IP does not define an application program interface, the standards do suggest the functionality needed. An interface must support the following conceptual operations:

---

[1] TLI stands for *Transport Layer Interface*.

- Allocate local resources for communication
- Specify local and remote communication endpoints
- Initiate a connection (client side)

- Wait for an incoming connection (server side)
- Send or receive data
- Determine when data arrives
- Generate urgent data
- Handle incoming urgent data
- Terminate a connection gracefully
- Handle connection termination from the remote site
- Abort communication
- Handle error conditions or a connection abort
- Release local resources when communication finishes

4.4 Conceptual Interface Specification

The TCP/IP standards do not leave implementors without any guidance. They specify a *conceptual interface* for TCP/IP that serves as an illustrative example. Because most operating systems use a procedural mechanism to transfer control from an application program into the system, the standard defines the conceptual interface as a set of procedures and functions. The standard suggests the parameters that each procedure or function requires as well as the semantics of the operation it performs. For example, the TCP standard discusses a *SEND* procedure, and lists the arguments an application needs to supply to send data on an existing TCP connection.

The point of defining conceptual operations is simple:

> The conceptual interface defined by the TCP/IP standards does not specify data representations or programming details; it merely provides an example of one possible interface that an operating system can offer to application programs that use TCP/IP

Thus, the conceptual interface illustrates loosely how applications interact with TCP. Because it does not prescribe exact details, operating system designers are free to choose alternative procedure names or parameters as long as they offer equivalent functionality.

4.5 System Calls

Figure 4.1 illustrates the *system call* mechanism that most operating systems use to transfer control between an application program and the operating system procedures that supply services. To a programmer, system calls look and act like function calls.

As the figure shows, when an application invokes a system call, control passes from the application to the system call interface. The interface then transfers control to the operating system. The operating system directs the incoming call to an internal procedure that performs the requested operation. Once the internal procedure completes, control returns through the system call interface to the application, which then continues to execute. In essence, whenever an application program needs

service from the operating system, the process executing the application climbs into the operating system, performs the necessary operation, and then climbs back out. As it passes through the system call interface, the process acquires privileges that allow it to read or modify data structures in the operating system. The operating system remains protected, however, because each system call branches to a procedure that the operating system designers have written.

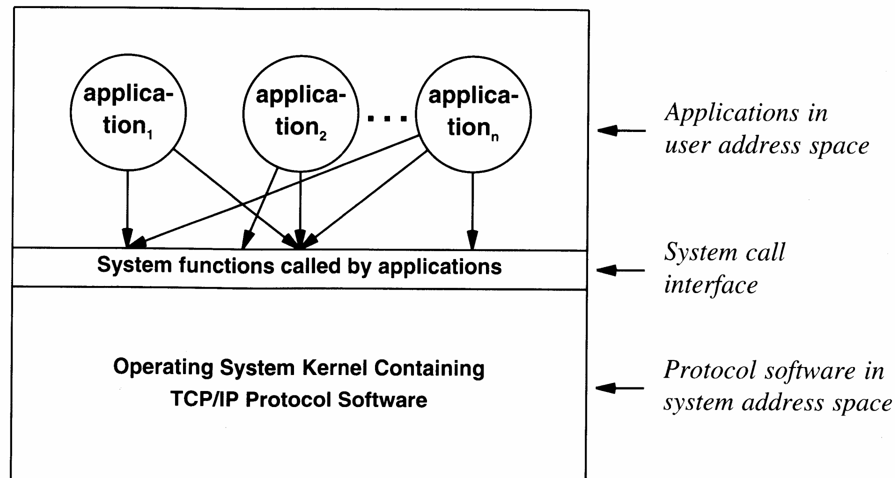

Figure 4.1 Applications interacting with TCP/IP protocol software through a system call interface. System calls behave like other function calls except that control transfers into the operating system.

4.6 Two Basic Approaches To Network Communication

Operating system designers must choose the exact set of procedures used to access TCP/IP protocols when they install protocol software in an operating system. Implementations follow one of two approaches:

- The designer invents entirely new system calls that applications use to access TCP/IP.
- The designer attempts to use conventional I/O calls to access TCP/IP.

In the first approach, the designer makes a list of all conceptual operations, invents names and parameters for each, and implements each as a system call. Because many designers consider it unwise to create new system calls unless absolutely necessary, this approach is seldom used. In the second approach, the designer uses conventional I/O primitives but overloads them so they work with network protocols as well as conventional I/O devices. Of course, many designers choose a hybrid approach that uses basic I/O functions whenever possible, but adds additional functions for those operations that cannot be expressed conveniently.

4.7 The Basic I/O Functions Available In UNIX

To understand how conventional system calls can be extended to accommodate TCP/IP consider the basic UNIX I/O functions. UNIX (and the many operating system variants derived from it) provides a basic set of six system functions used for input/output operations on devices or files. The table in Figure 4.2 lists the operations and their conventional meanings.

Operation	Meaning
open	**Prepare a device or a file for input or output operations**
close	**Terminate use of a previously opened device or file**
read	**Obtain data from an input device or file, and place it in the application program's memory**
write	**Transmit data from the application program's memory to an output device or file**
lseek	**Move to a specific position in a file or device (this operation only applies to files or devices like disks)**
ioctl†	**Control a device or the software used to access it (e.g., specify the size of a buffer or change the character set mapping)**

Figure 4.2 The basic I/O operations available in UNIX.

When an application program calls *open* to initiate input or output, the system returns a small integer called a *file descriptor* that the application uses in further I/O operations. The call to *open* takes three arguments: the name of a file or device to open, a set of bit flags that controls special cases such as whether to create the file if it does not exist, and an access mode that specifies read/write protections for newly created files. For example, the code segment:

```
int desc;

desc = open("filename", O_RDWR, 0)
```

opens an existing file, *filename,* with a mode that allows both reading and writing. After obtaining the integer descriptor, *desc,* the application uses it in further I/O operations on the file. For example, the statement:

```
read(desc, buffer, 128);
```

reads 128 bytes of data from the file into array *buffer.*

Finally, when an application finishes using a file, it calls *close* to deallocate the descriptor and release associated resources (e.g., internal buffers):

```
close(desc);
```

4.8 Using UNIX I/O With TCP/IP

When designers added TCP/IP protocols to UNIX, they extended the conventional UNIX I/O facilities. First, they extended the set of file descriptors and made it possible for applications to create descriptors used for network communication. Second, they extended the *read* and *write* system calls so they worked with the new network descriptors as well as with conventional file descriptors. Thus, when an application needs to send data across a TCP connection, it creates the appropriate descriptor, and then uses *write* to transfer data.

However, not all network communication fits easily into UNIX's *open- read-writeclose* paradigm. An application must specify the local and remote protocol ports and the remote IP address it will use, whether it will use TCP or UDP, and whether it will initiate transfer or wait for an incoming connection (i.e., whether it wants to behave as a client or server). If it is a server, it must specify how many incoming connection requests the operating system should enqueue before rejecting them. Furthermore,

if an application chooses to use UDP, it must be able to transfer UDP datagrams, not merely a stream of bytes. The designers of Berkeley UNIX added new system calls to UNIX to accommodate these special cases. The next chapter shows the details of the design.

4.9 Summary

Because TCP/IP is designed for a multi-vendor environment, the protocol standards loosely specify the interface that application programs use, allowing operating system designers freedom in choosing how to implement it. The standards do discuss a conceptual interface, but it is intended only as an illustrative example. Although the standards present the conceptual interface as a set of procedures, designers are free to choose different procedures or to use an entirely different style of interaction (e.g., message passing).

Operating systems often supply services through a mechanism known as the system call interface. When adding support for TCP/IP, designers attempt to minimize the number of new system calls by extending existing system calls where possible. However, because network communication requires operations that do not fit easily into conventional I/O procedures, most interfaces to TCP/IP require a few new system calls.

FOR FURTHER STUDY

Section 2 of the UNIX *Programmer's Manual* describes each of the socket calls in detail; section 4P describes protocols and network device interfaces in more detail. [AT&T 1989] defines AT&T's TLI interface, an alternative to sockets used in System V UNIX.

EXERCISES

4.1 Examine a message-passing operating system. How would you extend the application program interface to accommodate network communication?

4.2 Compare the socket interface from Berkeley UNIX with AT&T's TLI. What are the major differences? How are the two similar? What reasons could designers have for choosing one design over the other?

4.3 Some hardware architectures limit the number of possible system calls to a small number (e.g., 64 or 128). How many system calls have already been assigned in your local operating system?

4.4 Think about the hardware limit on system calls discussed in the previous exercise. How can a system designer add additional system calls without changing the hardware?

4.5 Find out how recent versions of the Kom shell use */dev/tcp* to allow UNIX shell scripts to communicate with TCP. Write an example script.

5
The Socket Interface

5.1 Introduction

The previous chapter describes the interface between application programs and the TCP/IP software, and shows how most systems use the system call mechanism to transfer control to the TCP/IP software in an operating system. It also reviews the six basic I/O functions that UNIX supplies: *open, close, read, write, lseek,* and *ioctl.* This chapter describes the details of a specific set of UNIX system calls for TCP/IP and discusses how they use the UNIX I/O paradigm. It covers concepts in general, and gives the intended use of each call. Later chapters show how clients and servers use these calls, and provide examples that illustrate many of the details.

5.2 Berkeley Sockets

In the early 1980s, the Advanced Research Projects Agency (ARPA) funded a group at the University of California at Berkeley to transport TCP/IP software to the UNIX operating system and to make the resulting software available to other sites. As part of the project, the designers created an interface that applications use to communicate. They decided to use the existing UNIX system calls whenever possible and to add new system calls to support TCP/IP functions that did not fit easily into the existing set of functions. The result became known as the *socket interface*[1], and the system is known as *Berkeley UNIX* or *BSD UNIX. (TCP* first appeared in release 4.1 of the Berkeley Software Distribution; the socket functions that this text describes are from release 4.4.)

Because many computer vendors, especially workstation manufacturers like Sun Microsystems Incorporated, Tektronix Incorporated, and Digital Equipment Corporation, adopted Berkeley UNIX, the socket interface has become available on many machines. Consequently, the socket interface has become so widely accepted that it ranks as a *de facto* standard.

5.3 Specifying A Protocol Interface

When designers consider how to add functions to an operating system that provide application programs access to TCP/IP protocol software, they must choose names for the functions and must specify the parameters that each function accepts. In so doing, they decide the scope of services that the functions supply and the style in which applications use them. Designers must also consider whether to make the interface specific to the TCP/IP protocols or whether to plan for additional protocols. Thus, the designers must choose one of two broad approaches:

- Define functions specifically to support TCP/IP communication.
- Define functions that support network communication in general, and use parameters to make TCP/IP communication a special case.

Differences between the two approaches are easiest to understand by their impact on the names of system functions and the parameters that the functions require. For example, in the first approach, a designer might choose to have a system function named *maketcpconnection,* while in the second, a designer might choose to create a general function *makeconnection* and use a parameter to specify the TCP protocol. Because the designers at Berkeley wanted to accommodate multiple sets of communication protocols, they used the second approach. In fact, throughout the design, they provided for generality far beyond TCP/IP. They allowed for multiple *families of* protocols, with all TCP/IP protocols represented as a single family (family *PF_INET).* They also decided to have applications specify operations using a *type of service* required instead of specifying the protocol name. Thus, instead of specifying that it wants a TCP connection, an application requests the *stream transfer* type of service using the Internet family *of* protocols. We can summarize:

---

[1] The socket interface is sometimes called the *Berkeley socket interface*.

The Berkeley socket interface provides generalized functions that support network communication using many possible protocols. Socket calls refer to all TCP/IP protocols as a single protocol family. The calls allow the programmer to specify the type of service required rather than the name of a specific protocol.

The overall design of sockets and the generality they provide have been debated since their inception. Some computer scientists argue that generality is unnecessary and merely makes application programs difficult to read. Others argue that having programmers specify the type of service instead of the specific protocol makes it easier to program because it frees the programmer from understanding the details of each protocol family. Finally, some commercial vendors of TCP/IP software have argued in favor of alternative interfaces because sockets cannot be added to an operating system unless the customer has the source code, which usually requires a special license agreement and additional expense.

5.4 The Socket Abstraction

5.4.1 Socket Descriptors And File Descriptors

In UNIX, an application that needs to perform I/O calls the *open* function to create a file descriptor that it uses to access the file. As Figure 5.1 shows, the operating system implements file descriptors as an array of pointers to internal data structures. The system maintains a separate file descriptor table for each process. When a process opens a file, the system places a pointer to the internal data structures for that file in the process' file descriptor table and returns the table index to the caller. The application program only needs to remember the descriptor and to use it in subsequent calls that request operations on the file. The operating system uses the descriptor as an index into the process' descriptor table, and follows the pointer to the data structures that hold all information about the file.

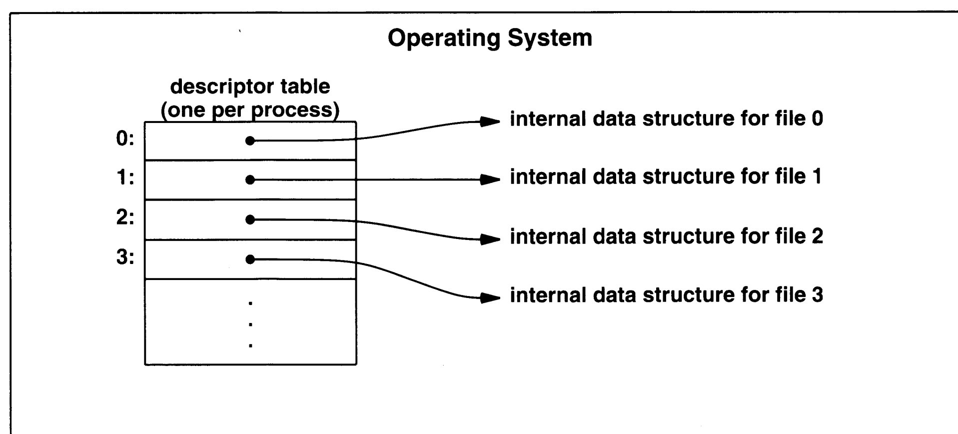

Figure 5.1 The per-process file descriptor table in UNIX. The operating system uses a process' descriptor table to store pointers to internal data structures for files that the process has opened. The process (application) uses the descriptor when referring to the file.

The socket interface adds a new abstraction for network communication, the *socket.* Like files, each active socket is identified by a small integer called its *socket descriptor. UNIX* allocates socket descriptors in the same descriptor table as file descriptors. Thus, an application cannot have both a file descriptor and a socket descriptor with the same value.

BSD UNIX contains a separate system function, *socket,* that applications call to create a socket; an application only uses *open* to create file descriptors.

The general idea underlying sockets is that a single system call is sufficient to create any socket. Once the socket has been created, an application must make additional system calls to specify the details of its exact use. The paradigm will become clear after we examine the data structures the system maintains.

5.4.2 System Data Structures For Sockets

The easiest way to understand the socket abstraction is to envision the data structures in the operating system. When an application calls *socket,* the operating system allocates a new data structure to hold the information needed for communication, and fills in a new descriptor table entry to contain a pointer to the data structure. For example, Figure 5.2 illustrates what happens to the descriptor table of Figure 5.1 after a call to *socket*[2]. In the example, arguments to the socket call have specified protocol family *PF_INET* and type of service *SOCK_STREAM*.

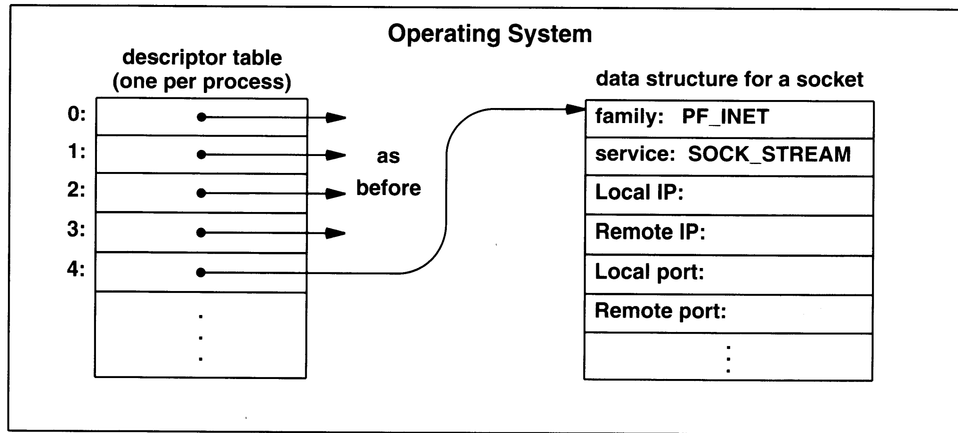

Figure 5.2 Conceptual operating system data structures after a call to *socket.* The system uses a single descriptor table for sockets and other I/O.

Although the internal data structure for a socket contains many fields, the system leaves most of them unfilled when it creates the socket. As we will see, the application that created the socket must make additional system calls to fill in information in the socket data structure before the socket can be used.

5.4.3 Using Sockets

Once a socket has been created, it can be used to wait for an incoming connection or to initiate a connection. A socket used by a server to wait for an incoming connection is called a *passive socket,* while a socket used by a client to initiate a connection is called an *active socket.* The only difference between active and passive sockets lies in how applications use them; the sockets are created the same way initially.

5.5 Specifying An Endpoint Address

When a socket is created, it does not contain detailed information about how it will be used. In particular, the socket does not contain information about the protocol port numbers or IP addresses of either the local machine or the remote machine. Before an application uses a socket, it must specify one or both of these addresses.

TCP/IP protocols define a *communication endpoint* to consist of an IP address and a protocol port number. Other protocol families define their endpoint addresses in other ways. Because the socket abstraction accommodates multiple families of protocols, it does not specify how to define endpoint addresses nor does it define a particular protocol address format. Instead, it allows each protocol family to specify endpoints however it likes.

To allow protocol families the freedom to choose representations for their addresses the socket abstraction defines an *address family* for each type of address. A protocol family can use one or more address families to define address representations. The TCP/IP protocols all use a single address representation, with the address family denoted by the symbolic constant AF_INET.

---

[2] UNIX data structures are more complex than shown in Figure 5.1; the diagram illustrates the concept, not the details.

In practice, much confusion arises between the TCP/IP protocol family, denoted *PF_INET,* and the address family it uses, denoted *AF_INET.* The chief problem is that both symbolic constants have the same numeric value (2), so programs that inadvertent ly use one in place of the other operate correctly. Even the Berkeley UNIX source code contains examples of misuse. Programmers should observe the distinction, however, because it helps clarify the meaning of variables and makes programs more portable.

5.6 A Generic Address Structure

Some software manipulates protocol addresses without knowing the details of how every protocol family defines its address representation. For example, it may be necessary to write a procedure that accepts an arbitrary protocol endpoint specification as an argument and chooses one of several possible actions depending on the address type. To accommodate such programs, the socket system defines a generalized format that all endpoint addresses use. The generalized format consists of a pair:

(address family, endpoint address in that family)

where the address family field contains a constant that denotes one of the preassigned address types, and the endpoint address field contains an endpoint address using the standard representation for the specified address type.

In practice, the socket software provides declarations of predefined C structures for address endpoints. Application programs use the predefined structures when they need to declare variables that store endpoint addresses or when they need to use an overlay to locate fields in a structure. The most general structure is known as a *sockaddr structure.* It contains a 2-byte address family identifier and a 14-byte array to hold an address[3]:

```
struct sockaddr {          /* struct to hold an address */
u_char sa_len;            /* total length */
u_short sa_family;        /* type of address */
char sa_data[14];         /* value of address */
```

Unfortunately, not all address families define endpoints that fit into the *sockaddr* structure. For example, BSD UNIX defines the *AF_UNIX* address family to specify what UNIX programmers think of as a named *pipe.* Endpoint addresses in the *AF UNIX* family consist of UNIX path names that can be much longer than 14 bytes. Therefore, application programs should not use *sockaddr_in* variable declarations because a variable declared to be of type *sockaddr* is not large enough to hold all possible endpoint addresses.

Confusion often arises in practice because the *sockaddr* structure accommodates addresses in the *AF_INET* family. Thus, TCP/IP software works correctly even if the programmer declares variables to be of type *sockaddr.* However, to keep programs portable and maintainable, TCP/IP code should not use the *sockaddr* structure in declarations. Instead, *sockaddr* should be used only as an overlay, and code should reference only the *sa_family* field in it.

Each protocol family that uses sockets defines the exact representation of its endpoint addresses, and the socket software provides corresponding structure declarations. Each TCP/IP endpoint address consists of a 2-byte field that identifies the address type (it must contain *AF_INET),* a 2-byte port number field, a 4-byte IP address field, and an 8-byte field that remains unused. Predefined structure *sockaddr_in* specifies the format:

```
struct sockaddr_in {                /* struct to hold an address */
    u_char sin_len;                 /* total length */
    u_short sin_family;             /* type of address */
    u_short sin_port;               /* protocol port number */
```

[3] This text describes the structure as defined in release 4.4 of the Berkeley software; older versions of the *sockaddr* structure do not include the *sa len* field.

```
    struct in_addr sin_addr;          /* IP address (declared to be   */
                                      / * u long on some systems)   * /
    char sin_zero[8];                 / * unused (set to zero) * /
};
```

An application that uses TCP/IP protocols exclusively can use structure *sockaddr_in* exclusively; it never needs to use the *sockaddr* structure[4]. Thus,

> *When representing a TCP/IP communication endpoint, an application program uses structure sockaddr_in, which contains both an IP address and a protocol port number. Programmers must be careful when writing programs that use a mixture of protocols because some non-TCP/IP endpoint addresses require a larger structure.*

5.7 Major System Calls Used With Sockets

Socket calls can be separated into two groups: primary calls that provide access to the underlying functionality and utility routines that help the programmer. This section describes the calls that provide the primary functionality that clients and servers need.

The details of socket system calls, their parameters, and their semantics can seem overwhelming. Much of the complexity arises because sockets have parameters that allow programs to use them in many ways. A socket can be used by a client or by a server, for stream transfer (TCP) or datagram (UDP) communication, with a specific remote endpoint address (usually needed by a client) or with an unspecified remote endpoint address (usually needed by a server).

To help understand sockets, we will begin by examining the primary socket calls and describing how a straightforward client and server use them to communicate with TCP. Later chapters each discuss one way to use sockets, and illustrate many of the details and subtleties not covered here.

5.7.1 The Socket Call

An application calls *socket* to create a new socket that can be used for network communication. The call returns a descriptor for the newly created socket. Arguments to the call specify the protocol family that the application will use (e.g., *PF_INET* for TCP/IP) and the protocol or type of service it needs (i.e., stream or datagram). For a socket that uses the Internet protocol family, the protocol or type of service argument determines whether the socket will use TCP or UDP.

5.7.2 The Connect Call

After creating a socket, a client calls *connect* to establish an active connection to a remote server. An argument to *connect* allows the client to specify the remote endpoint, which includes the remote machine's IP address and protocol port number. Once a connection has been made, a client can transfer data across it.

5.7.3 The Write Call

Both clients and servers use *write* to send data across a TCP connection. Clients usually use *write* to send requests, while servers use it to send replies. A call to *write* requires three arguments. The application passes the descriptor of a socket to which the data should be sent, the address of the data to be sent, and the length of the data. Usually, *write* copies outgoing data into buffers in the operating system kernel, and allows the application to continue execution while it transmits the data across the

---

[4] Structure *sockaddr* is used to cast (i.e., change the type of) pointers or the results of system functions to make programs pass the type checking in *lint*.

network. If the system buffers become full, the call to *write* may block temporarily until TCP can send data across the network and make space in the buffer for new data.

5.7.4 The Read Call

Both clients and servers use *read* to receive data from a TCP connection. Usually, after a connection has been established, the server uses *read* to receive a request that the client sends by calling *write*. After sending its request, the client uses *read* to receive a reply.

To read from a connection, an application calls *read* with three arguments. The first specifies the socket descriptor to use, the second specifies the address of a buffer, and the third specifies the length of the buffer. *Read* extracts data bytes that have ar rived at that socket, and copies them to the user's buffer area. If no data has arrived, the call to *read* blocks until it does. If more data has arrived than fits into the buffer, *read* only extracts enough to fill the buffer. If less data has arrived than fits into the buffer, *read* extracts all the data and returns the number of bytes it found.

Clients and servers can also use *read* to receive messages from sockets that use UDP. As with the connection-oriented case, the caller supplies three arguments that identify a socket descriptor, the address of a buffer into which the data should be placed, and the size of the buffer. Each call to *read* extracts one incoming UDP message (i.e., one user datagram). If the buffer cannot hold the entire message, *read* fills the buffer and discards the remainder.

5.7.5 The Close Call

Once a client or server finishes using a socket, it calls *close* to deallocate it. If only one process is using the socket, *close* immediately terminates the connection and deallocates the socket. If several processes share a socket, *close* decrements a reference count and deallocates the socket when the reference count reaches zero.

5.7.6 The Bind Call

When a socket is created, it does not have any notion of endpoint addresses (neither the local nor remote addresses are assigned). An application calls *bind* to specify the local endpoint address for a socket. The call takes arguments that specify a socket descriptor and an endpoint address. For TCP/IP protocols, the endpoint address uses the *sockaddr_in* structure, which includes both an IP address and a protocol port number. Primarily, servers use *bind* to specify the well-known port at which they will await connections.

5.7.7 The Listen Call

When a socket is created, the socket is neither *active (i.e.,* ready for use by a client) nor *passive (i.e.,* ready for use by a server) until the application takes further action. Connection-oriented servers call *listen* to place a socket in *passive mode* and make it ready to accept incoming connections.

Most servers consist of an infinite loop that accepts the next incoming connection, handles it, and then returns to accept the next connection. Even if handling a given connection takes only a few milliseconds, it may happen that a new connection request arrives during the time the server is busy handling an existing request. To ensure that no connection request is lost, a server must pass *listen* an argument that tells the operating system to enqueue connection requests for a socket. Thus, one argument to the *listen* call specifies a socket to be placed in passive mode, while the other specifies the size of the queue to be used for that socket.

5.7.8 The Accept Call

For TCP sockets, after a server calls *socket* to create a socket, *bind* to specify a local endpoint address, and *listen* to place it in passive mode, the server calls *accept* to extract the next incoming connection request. An argument to *accept* specifies the socket from which a connection should be accepted.

Accept creates a new socket for each new connection request, and returns the descriptor of the new socket to its caller. The server uses the new socket only for the new connection; it uses the original socket to accept additional connection requests. Once

it has accepted a connection, the server can transfer data on the new socket. After it finishes using the new socket, the server closes it.

5.7.9 Summary Of Socket Calls Used With TCP

The table in Figure 5.3 provides a brief summary of the system functions related to sockets.

Function Name	Meaning
socket	Create a descriptor for use in network communication
connect	Connect to a remote peer (client)
write	Send outgoing data across a connection
read	Acquire incoming data from a connection
close	Terminate communication and deallocate a descriptor
bind	Bind a local IP address and protocol port to a socket
listen	Place the socket in passive mode and set the number of incoming TCP connections the system will enqueue (server)
accept	Accept the next incoming connection (server)
recv	Receive the next incoming datagram
recvmsg	Receive the next incoming datagram (variation of recv)
recvfrom	Receive the next incoming datagram and record its source endpoint address
send	Send an outgoing datagram
sendmsg	Send an outgoing datagram (variation of send)
sendto	Send an outgoing datagram, usually to a prerecorded endpoint address
shutdown	Terminate a TCP connection in one or both directions
getpeername	After a connection arrives, obtain the remote machine's endpoint address from a socket
getsockopt	Obtain the current options for a socket
setsockopt	Change the options for a socket

Figure 5.3 A summary of the socket functions and the meaning of each.

5.8 Utility Routines For Integer Conversion

TCP/IP specifies a standard representation for binary integers used in protocol headers. The representation, known as *network byte order,* represents integers with the most significant byte first.

Although the protocol software hides most values used in headers from application programs, a programmer must be aware of the standard because some socket routines require arguments to be stored in network byte order. For example, the protocol port field of a *sockaddr_in* structure uses network byte order.

The socket routines include several functions that convert between network byte order and the local host's byte order. Programs should always call the conversion routines even if the local machine's byte order is the same as the network byte order because doing so makes the source code portable to an arbitrary architecture.

The conversion routines are divided into *short* and *long sets* to operate on 16-bit integers and 32-bit integers. Functions *htons (host to network short)* and *ntohs (network* to *host short)* convert a short integer from the host's native byte order to the network by to order, and vice versa. Similarly, *htonl* and *ntohl* convert long integers from the host's native byte order to network byte order and vice versa. To summarize:

Software that uses TCP/IP calls functions htons, ntohs, htonl and ntohl to convert binary integers between the host's native byte order and network standard byte order. Doing so makes the source code portable to any machine, regardless of its native byte order.

5.9 Using Socket Calls In A Program

Figure 5.4 illustrates a sequence of calls made by a client and a server using TCP.

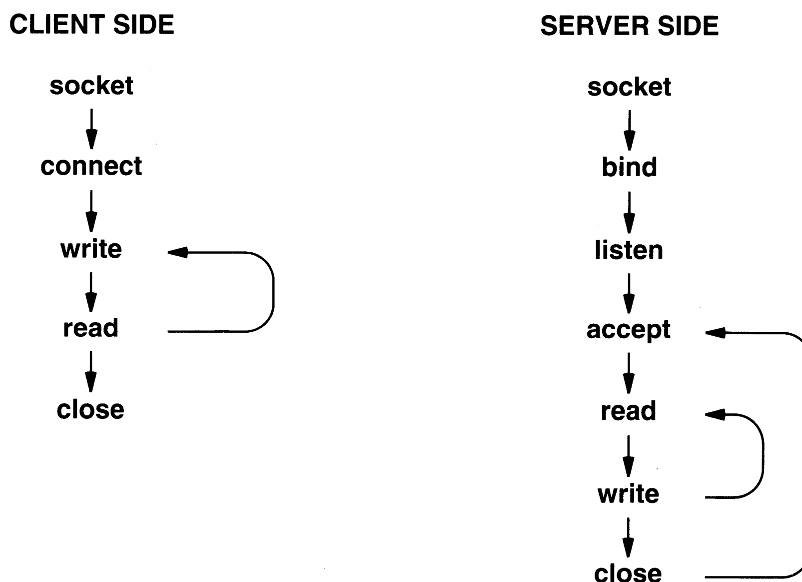

Figure 5.4 An example sequence of socket system calls made by a client and server using TCP. The server runs forever. It waits for a new connection on the well-known port, accepts the connection, interacts with the client, and then closes the connection.

The client creates a socket, calls *connect* to connect to the server, and then interacts using *write* to send requests and *read* to receive replies. When it finishes using the connection, it calls *close. A* server uses *bind* to specify the local (well-known) protocol port it will use, calls *listen* to set the length of the connection queue, and then enters a loop. Inside the loop, the server calls *accept* to wait until the next connection request arrives, uses *read* and *write* to interact with the client, and finally uses *close* to terminate the connection. The server then returns to the *accept* call, where it waits for the next connection.

5.10 Symbolic Constants For Socket Call Parameters

In addition to the system functions that implement sockets, BSD UNIX provides a set of predefined symbolic constants and data structure declarations that programs use to declare data and to specify arguments. For example, when specifying whether to use datagram service or stream service, an application program uses symbolic constants *SOCK DGRAM* or *SOCK STREAM. To* do so, the program must incorporate the appropriate definitions into each program with the C preprocessor *include* statement. Usually, *include* statements appear at the beginning of a source file; they must appear before any use of the constants they define. The *include* statements needed for sockets usually have the form:

```
#include <sys/types.h>
#include <sys/socket.h>
```

We will assume throughout the remainder of this text that programs always begin with these statements, even if they are not shown explicitly in the examples. To summarize:

UNIX supplies predefined symbolic constants and data structure declarations used with the socket system calls. Programs that reference these constants must begin with C preprocessor include statements that reference the files in which the definitions appear.

5.11 Summary

BSD UNIX introduced the socket abstraction as a mechanism that allows application programs to interface with protocol software in the operating system. Because so many other vendors have adopted sockets, they have become a *de facto* standard.

A program calls *socket* to create a socket and obtain a descriptor for it. Arguments to the *socket* call specify the protocol family to be used and the type of service required. All TCP/IP protocols are part of the Internet family, specified with symbolic constant *PF_INET.* The system creates an internal data structure for the socket, fills in the protocol family, and uses the type of service argument to select a specific protocol (usually either UDP or TCP).

Additional system calls allow the application to specify a local endpoint address*(bind),* to force the socket into passive mode for use by a server *(listen),* or to force the socket into active mode for use by a client *(connect).* Servers can make further calls to obtain incoming connection requests *(accept),* and both clients and servers can send or receive information *(read* and *write).* Finally, both clients and servers can deallocate a socket once they have finished using it *(close).*

The socket structure allows each protocol family to define one or more address representations. All TCP/IP protocols use the Internet address family, *AF_INET,* which specifies that an endpoint address contains both an IP address and a protocol port number. When an application specifies a communication endpoint to a socket function, it uses predefined structure *sockaddr_in. If* a client specifies that it needs an arbitrary, unused local protocol port, the TCP/IP software will select one.

Before an application program written in C can use the predefined structures and symbolic constants associated with sockets, it must include several files that define them. In particular, we assume that all source programs begin with statements that include files *<sys/types.h>* and *<sys/socket.h>*.

FOR FURTHER STUDY

Leffler et. al. [1989] describes the Berkeley UNIX system in detail, and describes the internal data structures UNIX uses for sockets. Presotto and Ritchie [June 1990] describes an interface for TCP/IP protocols using the UNIX file system space. The *UNIX Programmer's Manual* contains specifications for the socket functions, including an exact description of arguments and return codes. The section entitled *The IPC Tu*torial is worth reading. Much of the information on socket calls can also be found in Appendix A.

EXERCISES

5.1 Look at the *include* file for sockets (usually */usr/include/sys/socket.h).* What socket types are allowed? Which socket types do not makes sense for TCP/IP protocols?

5.2 If your system has a clock with at least microsecond accuracy, measure how long it takes to execute each of the socket system calls. Why do some calls require orders of magnitude more time than others?

5.3 Read the BSD UNIX manual pages for *connect* carefully. What network traffic is generated if one calls *connect* on a socket of type SOCK *DGRAM?*

5.4 Arrange to monitor your local network while an application executes *connect* for the first time on a socket of type SOCK *STREAM. How* many packets do you see?

6 Algorithms And Issues In Client Software Design

6.1 Introduction

Previous chapters consider the socket abstraction that applications use to interface with TCP/IP software, and review the system calls associated with it. This chapter discusses the basic algorithms underlying client software. It shows how applications become clients by initiating communication, how they use TCP or UDP protocols to contact a server, and how they use socket calls to interact with those protocols. The next chapter continues the discussion, and shows complete client programs that implement the ideas discussed here.

6.2 Learning Algorithms Instead Of Details

Because TCP/IP provides rich functionality that allows programs to communicate in a variety of ways, an application that uses TCP/IP must specify many details about the desired communication. For example, the application must specify whether it wishes to act as a client or a server, the endpoint address (or addresses) it will use, whether it will communicate with a connectionless or connection-oriented protocol, how it will enforce authorization and protection rules, and details such as the size of the buffers it will need.

So far, we have examined the set of operations available to an application without discussing how applications should use them. Unfortunately, knowing the low-level details of all possible system calls and their exact parameters does not provide programmers with an understanding of how to build well-designed, distributed programs. In fact, while a general understanding of the system calls used for network communication is important, few programmers remember all the details. Instead, they learn and remember the possible ways in which programs can interact across a network, and they understand the trade-offs of each possible design. In essence, programmers know enough about the algorithms underlying distributed computing to make design decisions and to choose among alternative algorithms quickly. They then consult a programming manual to find the details needed to write a program that implements a particular algorithm on a particular system. The point is that if the programmer knows *what* a program should do, finding out *how* to do it is straightforward.

> *Although programmers need to understand the conceptual capabilities of the protocol interface, they should concentrate on learning about ways to structure communicating programs instead of memorizing the details of a particular interface.*

6.3 Client Architecture

Applications that act as clients are conceptually simpler than applications that act as servers for several reasons. First, most client software does not explicitly handle concurrent interactions with multiple servers. Second, most client software executes as

a conventional application program. Unlike server software, client software does not usually require special privilege because it does not usually access privileged protocol ports. Third, most client software does not need to enforce protections. Instead, client programs can rely on the operating system to enforce protections automatically. In fact, designing and implementing client software is so straightforward that experienced application programmers can learn to write basic client applications quickly. The next sections discuss client software in general; later sections will focus on the differences between clients that use TCP and those that use UDP.

6.4 Identifying The Location Of A Server

Client software can use one of several methods to find a server's IP address and protocol port number. A client can:

- have the server's domain name or IP address specified as a constant when the program is compiled,
- require the user to identify the server when invoking the program,
- obtain information about the server from stable storage (e.g., from a file on a local disk), or
- use a separate protocol to find a server (e.g., multicast or broadcast a message to which all servers respond).

Specifying the server's address as a constant makes the client software faster and less dependent on a particular local computing environment. However, it also means that the client must be recompiled if the server is moved. More important, it means that the client cannot be used with an alternative server, even temporarily for testing. As a compromise, some clients fix a machine name instead of an IP address. Fixing the name instead of an address delays the binding until run-time. It allows a site to choose a generic name for the server and add an alias to the domain name system for that name. Using aliases permits a site manager to change the location of a server without changing client software. To move the server, the manager needs to change only the alias. For example, it is possible to add an alias for *mailhost* in the local domain and to arrange for all clients to look up the string "mailhost" instead of a specific machine. Because all clients reference the generic name instead of a specific machine, the system manager can change the location of the mail host without recompiling client software.

Storing the server's address in a file makes the client more flexible, but it means that the client program cannot execute unless the file is available. Thus, the client software cannot be transported to another machine easily.

While using a broadcast protocol to find servers works in a small, local environment, it does not scale well to large internets. Furthermore, use of a dynamic search mechanism introduces additional complexity for both clients and servers, and adds additional broadcast traffic to the network.

To avoid unnecessary complexity and dependence on the computing environment, most clients solve the problem of server specification in a simple manner: they require the user to supply an argument that identifies the server when invoking the client program. Building client software to accept the server address as an argument makes the client software general and eliminates dependency on the computing environment.

> *Allowing the user to specify a server address when invoking client software makes the client program more general and makes it possible to change server locations.*

An important point to note is that using an argument to specify the server's address results in the most flexibility. A program that accepts an address argument can be combined with other programs that extract the server address from disk, find the address using a remote nameserver, or search for it with a broadcast protocol. Thus,

> *Building client software that accepts a server address as an argument makes it easy to build extended versions of the software that use other ways to find the server address (e.g., read the address from a file on disk).*

Some services require an explicit server, while others can use any available server. For example, when a user invokes a remote login client, the user has a specific target machine in mind; logging into another machine usually does not make sense. However, if the user merely wants to find the current time of day, the user does not care which server responds. To accommodate such services, the designer can modify any of the server look up methods discussed above so they supply a set of server names instead of a single name. Clients must also be changed so they try each server in a set until they find one that responds.

6.5 Parsing An Address Argument

Usually, a user can specify arguments on the command line when invoking a client program. In most systems, each argument passed to a client program consists of a character string. The client uses an argument's syntax to interpret its meaning. For exam ple, most client software allows the user to supply either the domain name of the machine on which the server operates:

merlin.cd.purdue.edu

or an IP address in dotted decimal notation:

128.10.2.3

To determine whether the user has specified a name or an address, the client scans the argument to see if it contains alphabetic characters. If so, it must be a name. If it contains only digits and decimal points, the client assumes it to be a dotted decimal address and parses it accordingly.

Of course, client programs sometimes need additional information beyond the server's machine name or IP address. In particular, fully parameterized client software allows a user to specify a protocol port as well as a machine. It is possible to use an additional argument or to encode such information in a single string. For example, to specify the protocol port associated with the *smtp* service on machine with name *merlin.cs.purdue.edu,* the client could accept two arguments:

merlin.cs.purdue.edu smtp

or could combine both the machine name and protocol port into a single argument:

merlin. cs. purdue. edu: smtp

Although each client can choose the details of its argument syntax independently, having many clients with their own syntax can be confusing. From the user's point of view, consistency is always important. Thus, programmers are advised to follow whatever conventions their local system uses for client software. For example, because most existing UNIX software uses separate arguments to specify the server's machine and protocol port, new client software written for UNIX should use two arguments instead of one.

6.6 Looking Up A Domain Name

A client must specify the address of a server using structure *sockaddr_in.* Doing ;o means converting an address in dotted decimal notation (or a domain name in text form) into a 32-bit IP address represented in binary. Converting from dotted decimal notation to binary is trivial. Converting from a domain name, however, requires considerably more effort. The socket interface in BSD UNIX includes library routines, *inet addr* and *gethostbyname,* that perform the conversions. *Inet addr* takes an ASCII string that contains a dotted decimal address and returns the equivalent IP address in binary. *Gethostbyname* takes an ASCII string that contains the domain name for a machine. It returns the address of a *hostent* structure that contains, among other things, the host's IP address in binary. The *hostent* structure is declared in include file *netdb.h:*

```
struct hostent {
char      *h_name;            /* official host name */
char     **h_aliases;         /* other aliases      */
int        h_addrtype;        /* address type       */
int        h_length;          /* address length     */
char     **h_addr_list;       /* list of addresses  */
};
#define    h_addr h_addr_list[0]
```

Fields that contain names and addresses must be lists because hosts that have multiple interfaces also have multiple names and addresses. For compatibility with earlier versions, the file also defines the identifier *h addr* to refer to the first location in the host address list. Thus, a program can use *h addr* as if it were a field of the structure.

Consider a simple example of name conversion. Suppose a client has been passed the domain name *merlin.cs.purdue.edu* in string form and needs to obtain the IP address. The client can call *gethostbyname* as in:

```
struct hostent *hptr;
char *examplenam = "merlin.cs.purdue.edu";

if ( hptr = gethostbyname( examplenam ) ) {
    /* IP address is now in hptr->h_addr */
} else {
    /* error in name - handle it */
}
```

If the call is successful, *gethostbyname* returns a pointer to a valid *hostent* structure. If the name cannot be mapped into an IP address, the call returns zero. Thus, the client examines the value that *gethostbyname* returns to determine if an error occurred.

6.7 Looking Up A Well-Known Port By Name

Most client programs must look up the protocol port for the specific service they wish to invoke. For example, a client of an SMTP mail server needs to look up the well-known port assigned to SMTP. To do so, the client invokes library function *getservbyname,* which takes two arguments: a string that specifies the desired service and a string that specifies the protocol being used. It returns a pointer to a structure of type *servent,* also defined in include file *netdb.h:*

```
struct servent {
char *s_name;           /* official service name */
char **s_aliases;       /* other aliases         */
int sort;               /* port for this service */
char *s_proto;          /* protocol to use       */
```

If a TCP client needs to look up the official protocol port number for SMTP, it calls *getservbyname,* as in the following example:

```
struct servent *sptr;
if (sptr = getservbyname( "smtp", "tcp" )) {
      /* port number is now in sptr->s_port */
} else {
      /* error occurred - handle it */
}
```

6.8 Port Numbers And Network Byte Order

Function *getservbyname* returns the protocol port for the service in network byte order. Chapter 5 explains the concept of network byte order, and describes library routines that convert from network byte order to the byte order used on the local machine. It is sufficient to understand that *getservbyname* returns the port value in exactly the form needed for use in the *sockaddr_in* structure, but the representation may not agree with the local machine's usual representation. Thus, if a program prints out the value that *getservbyname* returns without converting to local byte order, it may appear to be incorrect.

6.9 Looking Up A Protocol By Name

The socket interface provides a mechanism that allows a client or server to map a protocol name to the integer constant assigned to that protocol. Library function *getprotobyname* performs the look up. A call passes the protocol name in a string argu ment, and *getprotobyname* returns the address of a structure of type *protoent.* If *getprotobyname* cannot access the database or if the specified name does not exist, it returns zero. The database of protocol names allows a site to define aliases for each name. The *protoent* structure has a field for the official protocol name as well as a field that points to the list of aliases. The C include file *netdb.* h contains the structure declaration:

```
struct protoent {
char  *p_name;        / * official protocol name  * /
char **p_aliases;     /* list of aliases allowed  */
int    p_proto;       /* official protocol number */
};
```

If a client needs to look up the official protocol number for UDP, it calls *getprotobyname, as* in the following example:

```
struct protoent *pptr;
if (pptr = getprotobyname ( "udp" )) {
   /* official protocol number is now in pptr->p _proto */
} else {
   /* error occurred - handle it */
}
```

6.10 The TCP Client Algorithm

Building client software is usually easier than building server software. Because TCP handles all reliability and flow control problems, building a client that uses TCP is the most straightforward of all network programming tasks. A TCP client follows Al gorithm 6.1 to form a connection to a server and communicate with it. The sections following the algorithm discuss each of its steps in more detail.

Algorithm 6.1

1. Find the IP address and protocol port number of the server with which communication is desired.
2. Allocate a socket.
3. Specify that the connection needs an arbitrary, unused protocol port on the local machine, and allow TCP to choose one.
4. Connect the socket to the server.
5. Communicate with the server using the application-level protocol (this usually involves sending requests and awaiting replies).
6. Close the connection.

Algorithm 6.1 A connection-oriented client. The client application allocates a socket and connects it to a server. It then sends requests across the connection and receives replies back.

6.11 Allocating A Socket

Previous sections have already discussed the methods used to find the server's IP address and the *socket* function used to allocate a communication socket. Clients that use TCP must specify protocol family *PF INET* and service SOCK *STREAM. A* pro gram begins with *include* statements that reference files which contain the definitions of symbolic constants used in the call and a declaration of the variable used to hold the socket descriptor. If more than one protocol in the family, specified by the first argument, offers the service requested by the second argument, the third argument to the *socket* call identifies a particular protocol. In the case of the Internet protocol family, only TCP offers the SOCK *STREAM* service. Thus, the third argument is irrelevant; zero should be used.

```
#include <sys/types.h>
#include <sys/socket.h>

int     s;            /* socket descriptor */

s = socket(PF_INET, SOCK_STREAM, 0);
```

6.12 Choosing A Local Protocol Port Number

An application needs to specify remote and local endpoint addresses for a socket before it can be used in communication. A server operates at a well-known protocol port address, which all clients must know. However, a TCP client does not operate on a preassigned port. Instead, it must select a local protocol port to use for its endpoint address. In general, the client does not care which port it uses as long as: (1) the port does not conflict with the ports that other processes on the machine are already using and (2) the port has not been assigned to a well-known service.

Of course, when a client needs a local protocol port, it could choose an arbitrary port at random until it finds one that meets the criteria given above. However, the socket interface makes choosing a client port much simpler because it provides a way that the client can allow TCP to choose a local port automatically. The choice of a local port that meets the criteria listed above happens as a side-effect of the *connect* call.

6.13 A Fundamental Problem In Choosing A Local IP Address

When forming a connection endpoint, a client must choose a local IP address as well as a local protocol port number. For a host that attaches to one network, the choice of a local IP address is trivial. However, because routers or multi-homed hosts have multiple IP addresses, making the choice can be difficult.

In general, the difficulty in choosing an IP address arises because the correct choice depends on routing and applications seldom have access to routing information. To understand why, imagine a computer with multiple network interfaces and, therefore, multiple IP addresses. Before an application can use TCP, it must have an endpoint address for the connection. When TCP communicates with a foreign destination, it encapsulates each TCP segment in an IP datagram and passes the datagram to the IP software. IP uses the remote destination address and its routing table to select a nexthop address and a network interface that it can use to reach the next hop.

Herein lies the problem: the IP source address in an outgoing datagram should match the IP address of the network interface over which IP routes the datagram. However, if an application chooses one of the machine's IP addresses at random, it might select an address that does not match that of the interface over which IP routes the traffic.

In practice, a client may appear to work even if the programmer chooses an incorrect address because packets may travel back to the client by a different route than they travel to the server. However, using an incorrect address violates the specification, makes network management difficult and confusing, and makes the program less reliable.

To solve the problem, the socket calls make it possible for an application to leave the local IP address field unfilled and to allow TCP/IP software to choose a local IP address automatically at the time the client connects to a server.

Because choosing the correct local IP address requires the application to interact with IP routing software, TCP client software usually leaves the local endpoint address unfilled, and allows TCP/IP software to select the correct local IP address and an unused local protocol port number automatically.

6.14 Connecting A TCP Socket To A Server

The *connect* system call allows a TCP client to initiate a connection. In terms of the underlying protocol, *connect* forces the initial TCP 3-way handshake. The call to *connect* does not return until a TCP connection has been established or TCP reaches a timeout threshold and gives up. The call returns 0 if the connection attempt succeeds or -1 if it fails. *Connect* takes three arguments:

```
retcode = connect(s , remaddr, remaddrlen)
```

where s is the descriptor for a socket, *remaddr* is the address of a structure of type *sockaddr_in* that specifies the remote endpoint to which a connection is desired, and *remaddrlen* is the length (in bytes) of the second argument.

Connect performs four tasks. First, it tests to ensure that the specified socket is valid and that it has not already been connected. Second, it fills in the remote endpoint address in the socket from the second argument. Third, it chooses a local endpoint ad dress for the connection (IP address and protocol port number) if the socket does not have one. Fourth, it initiates a TCP connection and returns a value to tell the caller whether the connection succeeded.

6.15 Communicating With The Server Using TCP

Assuming the *connect* call succeeds in establishing a connection, the client can use the connection to communicate with the server. Usually, the application protocol specifies a *request-response interaction* in which the client sends a sequence of *requests* and waits for a *response* to each.

Usually, a client calls *write* to transmit each request and *read* to await a response. For the simplest application protocols, the client sends only a single request and receives only a single response. More complicated application protocols require the client to iterate, sending a request and waiting for a response before sending the next request. The following code illustrates the request-response interaction by showing how a program writes a simple request over a TCP connection and reads a response:

```
/* Example code segment */
```

```
#define BLEN 120           /* buffer length to use */
char    *req = "request of some sort";
char     buf[BLEN] ;       /* buffer for answer    */
char    *bptr;             /* pointer to buffer    */
int      n;                /* number of bytes read */
int      buflen;           /* space left in buffer */

bptr = buf;
buf_len = BLEN;

/* send request */

write (s, req, strlen (req));

/* read response (may come in many pieces) */
while ((n = read(s, bptr, buflen) > 0) {
       bptr += n;
       buf_len -= n;
}
```

6.16 Reading A Response From A TCP Connection

The code in the previous example shows a client that sends a small message to a server and expects a small response (less than 120 bytes). The code contains a single call to *write,* but makes repeated calls to *read.* As long as the call to *read* returns data-," the code decrements the count of space available in the buffer and moves the buffer pointer forward past the data read. Iteration is necessary on input, even if the application at the other end of the connection sends only a small amount of data because TCP is not a block-oriented protocol. Instead, TCP is stream-oriented: it guarantees to deliver the sequence of bytes that the sender writes, but it does not guarantee to deliver them in the same grouping as they were written. TCP may choose to break a block of data into pieces and transmit each piece in a separate segment (e.g., it may choose to divide the data such that each piece fills the maximum sized segment, or it may need to send a small piece if the receiver does not have sufficient buffer space for a large one). Alternatively, TCP may choose to accumulate many bytes in its output buffer before sending a segment (e.g., to fill a datagram). As a result, the receiving application may receive data in small chunks, even if the sending application passes it to TCP in a single call to *write.* Or, the receiving application may receive data in a large chunk, even if the sending application passes it to TCP in a series of calls to *write.* The idea is fundamental to programming with TCP:

Because TCP does not preserve record boundaries, any program that reads from a TCP connection must be prepared to accept data a few bytes at a time. This rule holds even if the sending application writes data in large blocks.

6.17 Closing A TCP Connection 6.17.1 The Need For Partial Close

When an application finishes using a connection completely, it can call *close* to terminate the connection gracefully and deallocate the socket. However, closing a connection is seldom simple because TCP allows two-way communication. Thus, closing a connection usually requires coordination among the client and server.

To understand the problem, consider a client and server that use the requestresponse interaction described above. The client software repeatedly issues requests to which the server responds. On one hand, the server cannot terminate the connection be cause it cannot know whether the client will send additional requests. On the other hand, while the client knows when it has no more requests to send, it may not know whether all data has arrived from the server. The latter is especially important for application protocols that transfer arbitrary amounts of data in response to a request (e.g., the response to a database query).

6.17.2 A Partial Close Operation

To resolve the connection shutdown problem, most implementations of the socket interface include an additional primitive that permits applications to shut down a TCP connection in one direction. The *shutdown* system call takes two arguments, a socket descriptor and a direction specification, and shuts down the socket in the specified direction:

```
errcode = shutdown (s , direction);
```

The *direction* argument is an integer. If it contains 0, no further input is allowed. If it contains 1, no further output is allowed. Finally, if the value is 2, the connection is shutdown in both directions.

The advantage of a partial close should now be clear: when a client finishes sending requests, it can use *shutdown* to specify that it has no further data to send without deallocating the socket. The underlying protocol reports the shutdown to the remote machine, where the server application program receives an *end-of-file* signal. Once the server detects an end-of-file, it knows no more requests will arrive. After sending its last response, the server can close the connection. To summarize:

The partial close mechanism removes ambiguity for application protocols that transmit arbitrary amounts of information in response to a request. In such cases, the client issues a partial close after its last request; the server then closes the connection after its last response.

6.18 Programming A UDP Client

At first glance, programming a UDP client seems like an easy task. Algorithm 6.2 shows that the basic UDP client algorithm is similar to the client algorithm for TCP (Algorithm 6.1).

Algorithm 6.2

1. Find the IP address and protocol port number of the server with which communication is desired.
2. Allocate a socket.
3. Specify that the communication needs an arbitrary, unused protocol port on the local machine, and allow UDP to choose one.
4. Specify the server to which messages must be sent.
5. Communicate with the server using the application-level protocol (this usually involves sending requests and awaiting replies).
6. Close the socket.

Algorithm 6.2 A connectionless client. The sending process creates a connected socket and uses it to send one or more requests iteratively. This algorithm ignores the issue of reliability.

The first few steps of the UDP client algorithm are much like the corresponding steps of the TCP client algorithm. A UDP client obtains the server address and protocol port number, and then allocates a socket for communication.

6.19 Connected And Unconnected UDP Sockets

Client applications can use UDP in one of two basic modes: *connected* and *unconnected.* In connected mode, the client uses the *connect* call to specify a remote endpoint address (i.e., the server's IP address and protocol port number). Once it has specified the remote endpoint, the client can send and receive messages much like a TCP client does. In unconnected mode, the client does not connect the socket to a specific remote

endpoint. Instead, it specifies the remote destination each time it sends a message. The chief advantage of connected UDP sockets lies in their convenience for conventional client software that interacts with only one server at a time: the application only needs to specify the server once no matter how many datagrams it sends. The chief advantage of unconnected sockets lies in their flexibility; the client can wait to decide which server to contact until it has a request to send. Furthermore, the client can easily send each request to a different server.

UDP sockets can be connected, making it convenient to interact with a specific server, or they can be unconnected, making it necessary for the application to specify the server's address each time it sends a message.

6.20 Using Connect With UDP

Although a client can connect a socket of type *SOCK_DGRAM,* the *connect* call does not initiate any packet exchange, nor does it test the validity of the remote endpoint address. Instead, it merely records the remote endpoint information in the socket data structure for later use. Thus, when applied to SOCK*_DGRAM* sockets, *connect* only stores an address. Even if the *connect* call succeeds, it does not mean that the remote endpoint address is valid or that the server is reachable.

6.21 Communicating With A Server Using UDP

After a UDP client calls *connect,* it can use *write* to send a message or *read* to receive a response. Unlike TCP, UDP provides message transfer. Each time the client calls *write,* UDP sends a single message to the server. The message contains all the data passed to *write.* Similarly, each call to *read* returns one complete message. Assuming the client has specified a sufficiently large buffer, the *read* call returns all the data from the next message. Therefore, a UDP client does not need to make repeated calls to *read* to obtain a single message.

6.22 Closing A Socket That Uses UDP

A UDP client calls *close* to close a socket and release the resources associated with it. Once a socket has been closed, the UDP software will reject further messages that arrive addressed to the protocol port that the socket had allocated. However, the machine on which the *close* occurs does not inform the remote endpoint that the socket is closed. Therefore, an application that uses connectionless transport must be designed so the remote side knows how long to retain a socket before closing it.

6.23 Partial Close For UDP

Shutdown can be used with a connected UDP socket to stop further transmission in a given direction. Unfortunately, unlike the partial close on a TCP connection, when applied to a UDP socket, *shutdown* does not send any messages to the other side. In stead, it merely marks the local socket as unwilling to transfer data in the direction(s) >pecified. Thus, if a client shuts down further output on its socket, the server will not receive any indication that the communication has ceased.

6.24 A Warning About UDP Unreliability

Our simplistic UDP client algorithm ignores a fundamental aspect of UDP: namely, that it provides unreliable datagram delivery. While a simplistic UDP client can work well on local networks that exhibit low loss, low delay, and no packet reordering, clients that follow our algorithm will not work across a complex internet. To work in an internet environment, a client must implement reliability through timeout and retransmission. It must also handle the problems of duplicate or out-of-order packets. Adding reliability can be difficult, and requires expertise in protocol design.

> *Client software that uses UDP must implement reliability with techniques like packet sequencing, acknowledgements, timeouts, and retransmission. Designing protocols that are correct, reliable, and efficient for an internet environment requires considerable expertise.*

6.25 Summary

Client programs are among the most simple network programs. The client must obtain the server's IP address and protocol port number before it can communicate; to increase flexibility, client programs often require the user to identify the server when in voking the client. The client then converts the server's address from dotted decimal notation into binary, or uses the domain name system to convert from a textual machine name into an IP address.

The TCP client algorithm is straightforward: a TCP client allocates a socket and connects it to a server. The client uses *write* to send requests to the server and *read* to receive replies. Once it finishes using a connection, either the client or server invokes *close* to terminate it.

Although a client must explicitly specify the endpoint address of the server with which it wishes to communicate, it can allow TCP/IP software to choose an unused protocol port number and to fill in the correct local IP address. Doing so avoids the prob lem that can arise on a router or multi-homed host when a client inadvertently chooses an IP address that differs from the IP address of the interface over which IP routes the traffic.

The client uses *connect* to specify a remote endpoint address for a socket. When used with TCP, *connect* initiates a 3-way handshake and ensures that communication is possible. When used with UDP, *connect* merely records the server's endpoint address for later use.

Connection shutdown can be difficult if neither the client nor the server know exactly when communication has ended. To solve the problem, the socket interface supplies the *shutdown* primitive that causes a partial close and lets the other side know that no more data will arrive. A client uses *shutdown* to close the path leading to the server; the server receives an end-of-file signal on the connection that indicates the client has finished. After the server finishes sending its last response, it uses *close* to terminate the connection.

FOR FURTHER STUDY

Many RFCs that define protocols also suggest algorithms or implementation techniques for client code. Stevens [1990] also reviews client implementation.

EXERCISES

6.1 Read about the *sendto* and *recvfrom* socket calls. Do they work with sockets using TCP or sockets using UDP?

6.2 When the domain name system resolves a machine name, it returns a set of one or more IP addresses. Why?

6.3 Build client software that uses *gethostbyname* to look up machine names at your site and print all information returned. Which official names, if any, surprised you? Do you tend to use official machine names or aliases? Describe the circumstances, if any, when aliases may not work correctly.

6.4 Measure the time required to look up a machine name *(gethostbyname)* and a service entry *(getservent).* Repeat the test for both valid and invalid names. Does a look up for an invalid name take substantially longer than for a valid one? Explain any differences you observe.

6.5 Use a network monitor to watch the network traffic your computer generates when you look up an IP address name using *gethostbyname.* Run the experiment more than one time for each machine name you resolve. Explain the differences in network traffic between look ups.

6.6 To test whether your machine's local byte order is the same as the network byte order, write a program that uses *getservbyname* to look up the *ECHO* service for UDP and then prints the resulting protocol port value. If the local byte order and network byte order agree, the value will be 7.

6.7 Write a program that allocates a local protocol port, closes the socket, delays a few seconds, and allocates another local port. Run the program on an idle machine and on a busy timesharing system. Which port values did your program receive on each system? If they are not the same, explain.

6.8 Under what circumstances can a client program use close instead *of shutdown?*

6.9 Should a client use the same protocol port number each time it begins? Why or why not?

7
Example Client Software

7.1 Introduction

The previous chapter discusses the basic algorithms underlying client applications as well as specific techniques used to implement those algorithms. This chapter gives examples of complete, working client programs that illustrate the concepts in more de tail. The examples use UDP as well as TCP. Most important, the chapter shows how a programmer can build a library of procedures that hide the details of socket calls and make it easier to construct client software that is portable and maintainable.

7.2 The Importance Of Small Examples

TCP/IP defines a myriad of services and the standard application protocols for accessing them. The services range in complexity from the trivial (e.g., a character generator service used only for testing protocol software) to the complex (e.g., a file transfer service that provides authentication and protection). The examples in this chapter and the next few chapters concentrate on implementations of client-server software for simple services. Later chapters review client-server applications for several of the complex services.

While it may seem that the protocols used in the examples do not offer interesting or useful services, studying them is important. First, because the services themselves require little code, the client and server software that implements them is easy to under stand. More important, the small program size highlights fundamental algorithms and illustrates clearly how client and server programs use system functions. Second, studying simple services provides the reader with an intuition about the relative size of services and the number of services available. Having an intuitive understanding of small services will be especially important for the chapters that discuss the need for multiprotocol and multiservice designs.

7.3 Hiding Details

Most programmers understand the advantage of dividing large, complex programs into a set of procedures: a modular program becomes easier to understand, debug, and modify than an equivalent monolithic program. If programmers design procedures care fully, they can reuse them in other programs. Finally, choosing procedures carefully can also make a program easier to port to new computer systems.

Conceptually, procedures raise the level of the language that programmers use by hiding details. Programmers working with the low-level facilities available in most programming languages find programming tedious and prone to error. They also find themselves repeating basic segments of code in each program they write. Using procedures helps avoid repetition by providing higher-level operations. Once a particular algorithm has been encoded in a procedure, the programmer can use it in many programs without having to consider the implementation details again.

A careful use of procedures is especially important when building client and server programs. First, because network software includes declarations for items like endpoint addresses, building programs that use network services involves a myriad of tedious de tails not found in conventional programs. Using procedures to hide those details reduces the chance for error. Second, much of the code needed to allocate a socket, bind addresses, and form a network connection is repeated in each client; placing it in procedures allows programmers to reuse the code instead of replicating it. Third, because TCP/IP was designed to interconnect heterogeneous machines, network applications often operate on many different machine architectures. Programmers can use procedures to isolate operating system dependencies, making it easier to port code to a new machine.

7.4 An Example Procedure Library For Client Programs

To understand how procedures can make the programming task easier, consider the problem of building client programs. To establish connectivity with a server, a client must choose a protocol (like TCP or UDP), look up the server's machine name, look up and map the desired service into a protocol port number, allocate a socket, and connect it. Writing the code for each of these steps from scratch for each application wastes time. Furthermore, if programmers ever need to change any of the details, they have to modify each application. To minimize programming time, a programmer can write the code once, place it in a procedure, and simply call the procedure from each client program.

The first step of designing a procedure library is abstraction: a programmer must imagine high-level operations that would make writing programs simpler. For example, an application programmer might imagine two procedures that handle the work of allocating and connecting a socket:

socket = connectTCP(*machine, service);*

and

socket = connectUDP(*machine, service);*

It is important to understand that this is not a prescription for the "right" set of abstractions, it merely gives one possible way to form such a set. The important idea is:

> *The procedural abstraction allows programmers to define high-level operations, share code among applications, and reduce the chances of making mistakes with small details. Our example procedures used throughout this text merely illustrate one possible approach; programmers should feel free to choose their own abstractions.*

7.5 Implementation Of ConnectTCP

Because both of the proposed procedures, *connectTCP* and *connectUDP,* need to allocate a socket and fill in basic information, we chose to place all the low-level code in a third procedure, *connectsock,* and to implement both higher-level operations as simple calls. File *connectTCP. c* illustrates the concept:

```
/* connectTCP.c - connectTCP */


int connectsock(const char *host, const char *service,

        const char *transport);


/*------------------------------------------------------------------------

 * connectTCP - connect to a specified TCP service on a specified host

 *------------------------------------------------------------------------

 */

int

connectTCP(const char *host, const char *service )

/*

 * Arguments:

 *      host    - name of host to which connection is desired

 *      service - service associated with the desired port

 */

{

    return connectsock( host, service, "tcp");
```

```
}
```

7.6 Implementation Of ConnectUDP

File *connectUDP. c* shows how *connectsock* can be used to establish a connected socket that uses UDP.

```
/* connectUDP.c - connectUDP */

int connectsock(const char *host, const char *service,
        const char *transport);

/*------------------------------------------------------------------------
 * connectUDP - connect to a specified UDP service on a specified host
 *------------------------------------------------------------------------
 */
int
connectUDP(const char *host, const char *service )
/*
 * Arguments:
 *      host    - name of host to which connection is desired
 *      service - service associated with the desired port
 */
{
    return connectsock(host, service, "udp");
}
```

7.7 A Procedure That Forms Connections

Procedure *connectsock* contains all the code needed to allocate a socket and connect it. The caller specifies whether to create a UDP socket or a TCP socket.

```
/* connectsock.c - connectsock */

#define __USE_BSD   1

#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>
#include <string.h>
#include <stdlib.h>
```

```
#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff
#endif  /* INADDR_NONE */

typedef unsigned short u_short;
extern int  errno;

int errexit(const char *format, ...);

/*------------------------------------------------------------------------
 * connectsock - allocate & connect a socket using TCP or UDP
 *------------------------------------------------------------------------
 */
int
connectsock(const char *host, const char *service, const char *transport )
/*
 * Arguments:
 *      host      - name of host to which connection is desired
 *      service   - service associated with the desired port
 *      transport - name of transport protocol to use ("tcp" or "udp")
 */
{
    struct hostent  *phe;   /* pointer to host information entry    */
    struct servent  *pse;   /* pointer to service information entry */
    struct protoent *ppe;   /* pointer to protocol information entry*/
    struct sockaddr_in sin; /* an Internet endpoint address     */
    int s, type;    /* socket descriptor and socket type    */



    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;

    /* Map service name to port number */
    if ( pse = getservbyname(service, transport) )
        sin.sin_port = pse->s_port;
    else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
        errexit("can't get \"%s\" service entry\n", service);

    /* Map host name to IP address, allowing for dotted decimal */
    if ( phe = gethostbyname(host) )
        memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
```

```
    else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
        errexit("can't get \"%s\" host entry\n", host);

    /* Map transport protocol name to protocol number */
    if ( (ppe = getprotobyname(transport)) == 0)
        errexit("can't get \"%s\" protocol entry\n", transport);

    /* Use protocol to choose a socket type */
    if (strcmp(transport, "udp") == 0)
        type = SOCK_DGRAM;
    else
        type = SOCK_STREAM;

    /* Allocate a socket */
    s = socket(PF_INET, type, ppe->p_proto);
    if (s < 0)
        errexit("can't create socket: %s\n", strerror(errno));

    /* Connect the socket */
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't connect to %s.%s: %s\n", host, service,
            strerror(errno));
    return s;
}
```

Although most steps are straightforward, a few details make the code seem complicated. First, the C language permits complex expressions. As a result, the expressions in many of the condition statements contain a function call, an assignment, and a com parison, all on one line. For example, the call to *getprotobyname* appears in an expression that assigns the result to variable *ppe,* and then compares the result to 0. If the value returned is zero (i.e., an error occurred), the if statement executes a call to *errexit.*

Otherwise, the procedure continues execution. Second, the code uses two library procedures defined by ANSI *C, memset* and *memcpy*[1]. Procedure *memset* places bytes of a given value in a block of memory; it is the fastest way to zero a large structure or array. Procedure *memcpy* copies a block of bytes from one memory location to another, regardless of the contents[2]. *Connectsock* uses *memset* to fill the entire *sockaddr_in* structure with zeroes, and then uses *memcpy* to copy the bytes of the server's IP address into field *sin addr.* Finally, *Connectsock* calls procedure *connect* to connect the socket. If an error occurs, it calls *errexit.*

```
/* errexit.c - errexit */

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
```

---

[1] Early versions of UNIX used the names *bzero* and *bcopy*.

[2] Function *strcpy* cannot be used to copy an IP address because IP addresses can contain zero bytes which *strcpy* interprets as *end of string*.

```
/*------------------------------------------------------------------------
 * errexit - print an error message and exit
 *------------------------------------------------------------------------
 */
/*VARARGS1*/
int
errexit(const char *format, ...)
{
    va_list args;

    va_start(args, format);
    vfprintf(stderr, format, args);
    va_end(args);
    exit(1);
}
```

Errexit takes a variable number of arguments, which it passes on to *vfprintf* for output. *Errexit* follows the *printf* conventions for formatted output. The first argument specifies how the output should be formatted; remaining arguments specify values to be printed according to the given format.

7.8 Using The Example Library

Once programmers have selected abstractions and built a library of procedures, they can construct client applications. If the abstractions have been selected well, they make application programming simple and hide many of the details. To illustrate how our example library works, we will use it to construct example client applications. Because the clients each access one of the standard TCP/IP services, they also serve to illustrate several of the simpler application protocols.

7.9 The DAYTIME Service

The TCP/IP standards define an application protocol that allows a user to obtain the date and time of day in a format fit for human consumption. The service is officially named the *DAYTIME service*.

To access the DAYTIME service, the user invokes a client application. The client contacts a server to obtain the information, and then prints it. Although the standard does not specify the exact syntax, it suggests several possible formats. For example, DAYTIME could supply a date in the form:

weekday, month day, year time-timezone

like

Thursday, February 22, 1996 17:37:43-PST

The standard specifies that DAYTIME is available for both TCP and UDP. In both cases, it operates at protocol port 13.

The TCP version of DAYTIME uses the presence of a TCP connection to trigger output: as soon as a new connection arrives, the server forms a text string that contains the current date and time, sends the string, and then closes the connection. Thus, the client need not send any request at all. In fact, the standard specifies that the server must discard any data sent by the client.

The UDP version of DAYTIME requires the client to send a request. A request consists of an arbitrary UDP datagram. Whenever a server receives a datagram, it formats the current date and time, places the resulting string in an outgoing datagram, and sends it back to the client. Once it has sent a reply, the server discards the datagram that triggered the response.

7.10 Implementation Of A TCP Client For DAYTIME

File *TCPdaytime.c* contains code for a TCP client that accesses the DAYTIME service.

```
/* TCPdaytime.c - TCPdaytime, main */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int  errno;

int TCPdaytime(const char *host, const char *service);
int errexit(const char *format, ...);
int connectTCP(const char *host, const char *service);

#define LINELEN     128

/*------------------------------------------------------------------------
 * main - TCP client for DAYTIME service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *host = "localhost";    /* host to use if none supplied */
    char    *service = "daytime";   /* default service port     */

    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        fprintf(stderr, "usage: TCPdaytime [host [port]]\n");
        exit(1);
    }
```

```
    TCPdaytime(host, service);
    exit(0);
}


/*------------------------------------------------------------------------
 * TCPdaytime - invoke Daytime on specified host and print results
 *------------------------------------------------------------------------
 */
TCPdaytime(const char *host, const char *service)
{
    char    buf[LINELEN+1];     /* buffer for one line of text  */
    int s, n;           /* socket, read count       */

    s = connectTCP(host, service);

    while( (n = read(s, buf, LINELEN)) > 0) {
        buf[n] = '\0';      /* ensure null-terminated   */
        (void) fputs( buf, stdout );
    }
}
```

Notice how using *connectTCP* simplifies the code. Once a connection has been established, DAYTIME merely reads input from the connection and prints it, iterating until it detects an end of file condition.

7.11 Reading From A TCP Connection

The DAYTIME example illustrates an important idea: TCP offers a stream service that does not guarantee to preserve record boundaries. In practice, the stream paradigm means that TCP decouples the sending and receiving applications. For example, sup pose the sending application transfers 64 bytes of data in a single call to *write,* followed by 64 bytes in a second call. The receiving application may receive all 128 bytes in a single call to *read,* or it may receive 10 bytes in the first call, 100 bytes in the second call, and 18 bytes in a third call. The number of bytes returned in a call depends on the size of datagrams in the underlying internet, the buffer space available, and the delays encountered when crossing the internet.

> *Because the TCP stream service does not guarantee to deliver data in the same blocks that it was written, an application receiving data from a TCP connection cannot depend on all data being delivered in a single transfer; it must repeatedly call read until all data has been obtained.*

7.12 The TIME Service

TCP/IP defines a service that allows one machine to obtain the current date and time of day from another. Officially named *TIME,* the service is quite simple: a client program executing on one machine sends a request to a server executing on another. Whenever the server receives a request, it obtains the current date and time of day from the local operating system, encodes the information in a standard format, and sends it back to the client in a response.

To avoid the problems that occur if the client and server reside in different timezones, the TIME protocol specifies that all time and date information must be represented in *Universal Coordinated Time*[3], abbreviated UCT or UT. Thus, a server

---

[3] Universal Coordinated Time was formerly known as *Greenwich Mean Time*.

converts from its local time to universal time before sending a reply, and a client converts from universal time to its local time when the reply arrives.

Unlike the DAYTIME service, which is intended for human users, the TIME service is intended for use by programs that store or manipulate times. The TIME protocol always specifies time in a 32-bit integer, representing the number of seconds since an *epoch date*. The TIME protocol uses midnight, January l, 1900, as its epoch.

Using an integer representation allows computers to transfer time from one machine to another quickly, without waiting to convert it into a text string and back into an integer. Thus, the TIME service makes it possible for one computer to set its timeof-day clock from the clock on another system.

7.13 Accessing The TIME Service

Clients can use either TCP or UDP to access the TIME service at protocol port 37 technically, the standards define two separate services, one for UDP and one for TCP). A TIME server built for TCP uses the presence of a connection to trigger output, much like the DAYTIME service discussed above. The client forms a TCP connection to a TIME server and waits to read output. When the server detects a new connection, it sends the current time encoded as an integer, and then closes the connection. The client does not send any data because the server never reads from the connection.

Clients can also access a TIME service with UDP. To do so, a client sends a request, which consists of a single datagram. The server does not process the incoming datagram, except to extract the sender's address and protocol port number for use in a reply. The server encodes the current time as an integer, places it in a datagram, and sends the datagram back to the client.

7.14 Accurate Times And Network Delays

Although the TIME service accommodates differences in timezones, it does not handle the problem of network latency. If it takes 3 seconds for a message to travel from the server to the client, the client will receive a time that is 3 seconds behind that of the server. Other, more complex protocols handle clock synchronization. However, the TIME service remains popular for three reasons. First, TIME is extremely simple compared to clock synchronization protocols. Second, most clients contact servers on a local area network, where network latency accounts for only a few milliseconds. Third, except when using programs that use timestamps to control processing, humans do not care if the clocks on their computers differ by small amounts.

In cases where more accuracy is required, it is possible to improve TIME or use an alternative protocol. The easiest way to improve the accuracy of TIME is to compute an approximation of network delay between the server and client, and then add that ap proximation to the time value that the server reports. For example, one way to approximate latency requires the client to compute the time that elapses during the round trip from client to server and back. The client assumes equal delay in both directions, and obtains an approximation for the trip back by dividing the round trip time in half. It adds the delay approximation to the time of day that the server returns.

7.15 A UDP Client For The TIME Service

File *UDPtime.c* contains code that implements a UDP client for the TIME service.

```
/* UDPtime.c - main */


#include <sys/types.h>


#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```
#define BUFSIZE 64

#define UNIXEPOCH   2208988800  /* UNIX epoch, in UCT secs  */
#define MSG     "what time is it?\n"

typedef unsigned long u_long;
extern int  errno;

int connectUDP(const char *host, const char *service);
int errexit(const char *format, ...);

/*------------------------------------------------------------------------
 * main - UDP client for TIME service that prints the resulting time
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *host = "localhost";    /* host to use if none supplied */
    char    *service = "time";  /* default service name     */
    time_t  now;            /* 32-bit integer to hold time  */
    int s, n;           /* socket descriptor, read count*/

    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        fprintf(stderr, "usage: UDPtime [host [port]]\n");
        exit(1);
    }

    s = connectUDP(host, service);

    (void) write(s, MSG, strlen(MSG));
```

```
    /* Read the time */

    n = read(s, (char *)&now, sizeof(now));
    if (n < 0)
        errexit("read failed: %s\n", strerror(errno));
    now = ntohl((u_long)now);   /* put in host byte order   */
    now -= UNIXEPOCH;       /* convert UCT to UNIX epoch    */
    printf("%s", ctime(&now));
    exit(0);
}
```

The example code contacts the TIME service by sending a datagram. It then calls *read* to wait for a reply and extract the time value from it. Once *UDPtime* has obtained the time, it must convert the time into a form suitable for the local machine. First, it uses *ntohl* to convert the 32-bit value (a *long* in C) from network standard byte order into the local host byte order. Second, *UDPtime* must convert to the machine's local representation. The example code is designed for UNIX. Like the Internet protocols, UNIX represents time in a 32-bit integer and interprets the integer to be a count of seconds. Unlike the Internet, however, UNIX assumes an epoch date of January 1, 1970. Thus, to convert from the TIME protocol epoch to the UNIX epoch, the client must subtract the number of seconds between January 1, 1900 and January 1, 1970. The example code uses the conversion value 2208988800. Once the time has been converted to a representation compatible with that of the local machine, *UDPtime* can invoke the library procedure *ctime,* which converts the value into a human readable form for output.

7.16 The ECHO Service

TCP/IP standards specify an *ECHO service* for both UDP and TCP protocols. At first glance, ECHO services seem almost useless because an ECHO server merely returns all the data it receives from a client. Despite their simplicity, ECHO services are important tools that network managers use to test reachability, debug protocol software, and identify routing problems.

The TCP ECHO service specifies that a server must accept incoming connection requests, read data from the connection, and write the data back over the connection until the client terminates the transfer. Meanwhile, the client sends input and then reads it back.

7.17 A TCP Client For The ECHO Service

File *TCPecho. c* contains a simple client for the ECHO service.

```
/* TCPecho.c - main, TCPecho */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int  errno;

int TCPecho(const char *host, const char *service);
int errexit(const char *format, ...);
int connectTCP(const char *host, const char *service);
```

```
#define LINELEN     128

/*------------------------------------------------------------------------
 * main - TCP client for ECHO service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *host = "localhost";    /* host to use if none supplied */
    char    *service = "echo";  /* default service name     */

    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        fprintf(stderr, "usage: TCPecho [host [port]]\n");
        exit(1);
    }
    TCPecho(host, service);
    exit(0);
}

/*------------------------------------------------------------------------
 * TCPecho - send input to ECHO service on specified host and print reply
 *------------------------------------------------------------------------
 */
int
TCPecho(const char *host, const char *service)
{
    char    buf[LINELEN+1];     /* buffer for one line of text  */
    int s, n;           /* socket descriptor, read count*/
    int outchars, inchars;  /* characters sent and received */
```

```
    s = connectTCP(host, service);

    while (fgets(buf, sizeof(buf), stdin)) {
        buf[LINELEN] = '\0';    /* insure line null-terminated  */
        outchars = strlen(buf);
        (void) write(s, buf, outchars);

        /* read it back */
        for (inchars = 0; inchars < outchars; inchars+=n ) {
            n = read(s, &buf[inchars], outchars - inchars);
            if (n < 0)
                errexit("socket read failed: %s\n",
                    strerror(errno));
        }
        fputs(buf, stdout);
    }
}
```

After opening a connection, *TCPecho* enters a loop that repeatedly reads one line of input, sends the line across the TCP connection to the ECHO server, reads it back again, and prints it. After all input lines have been sent to the server, received back, and printed successfully, the client exits.

7.18 A UDP Client For The ECHO Service

File *UDPecho. c* shows how a client uses UDP to access an ECHO service.

```
/* UDPecho.c - main, UDPecho */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int  errno;

int UDPecho(const char *host, const char *service);
int errexit(const char *format, ...);
int connectUDP(const char *host, const char *service);

#define LINELEN     128

/*------------------------------------------------------------------------
 * main - UDP client for ECHO service
```

```
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *host = "localhost";
    char    *service = "echo";

    switch (argc) {
    case 1:
        host = "localhost";
        break;
    case 3:
        service = argv[2];
        /* FALL THROUGH */
    case 2:
        host = argv[1];
        break;
    default:
        fprintf(stderr, "usage: UDPecho [host [port]]\n");
        exit(1);
    }
    UDPecho(host, service);
    exit(0);
}

/*------------------------------------------------------------------------
 * UDPecho - send input to ECHO service on specified host and print reply
 *------------------------------------------------------------------------
 */
int
UDPecho(const char *host, const char *service)
{
    char    buf[LINELEN+1];     /* buffer for one line of text  */
    int s, nchars;      /* socket descriptor, read count*/

    s = connectUDP(host, service);

    while (fgets(buf, sizeof(buf), stdin)) {
        buf[LINELEN] = '\0';    /* insure null-terminated */
        nchars = strlen(buf);
```

```
            (void) write(s, buf, nchars);

            if (read(s, buf, nchars) < 0)
                errexit("socket read failed: %s\n",
                        strerror(errno));
            fputs(buf, stdout);
        }
}
```

The example UDP ECHO client follows the same general algorithm as the TCP version. It repeatedly reads a line of input, sends it to the server, reads it back from the server, and prints it. The biggest difference between the UDP and TCP versions lies in how they treat data received from the server. Because UDP is datagram-oriented, the client treats an input line as a unit and places each in a single datagram. Similarly, the ECHO server receives and returns complete datagrams. Thus, while the TCP client reads incoming data as a stream of bytes, the UDP client either receives an entire line back from the server or receives none of it; each call to *read* returns the entire line unless an error has occurred.

7.19 Summary

Programmers use the procedural abstraction to keep programs flexible and easy to maintain, to hide details, and to make it easy to port programs to new computers. Once a programmer writes and debugs a procedure, he or she places it in a library where it can be reused in many programs easily. A library of procedures is especially important for programs that use TCP/IP because they often operate on multiple computers.

This chapter presents an example library of procedures used to create client software. The primary procedures in our library, *connectTCP* and *connectUDP,* make it easy to allocate and connect a socket to a specified service on a specified host.

The chapter presents examples of a few client applications. Each example contains the code for a complete C program that implements a standard application protocol: DAYTIME (used to obtain and print the time of day in a human-readable format), TIME (used to obtain the time in 32-bit integer form), and ECHO (used to test network connectivity). The example code shows how a library of procedures hides many of the details associated with socket allocation and makes it easier to write client software.

FOR FURTHER STUDY

The application protocols described here are each part of the TCP/IP standard. Postel [RFC 867] contains the standard for the DAYTIME protocol, Postel and Harrenstien [RFC 868] contains the standard for the TIME protocol, and Postel [RFC 862] contains the standard for the ECHO protocol. Mills [RFC 1305] specifies version 3 of the Network Time Protocol, NTP.

EXERCISES

7.1 Use program *TCPdaytime* to contact servers on several machines. How does each format the time and date?

7.2 The Internet standard represents time in a 32-bit integer that gives seconds past the epoch, midnight January 1, 1900. UNIX systems also represent time in a 32-bit integer that measures seconds, but UNIX uses January 1, 1970 as its epoch. What is the maximum date and time that can be represented in each system?

7.3 Improve the TIME client so it checks the date received to verify that it is greater than January 1, 1996 (or some other date you know to be in the recent past).

7.4 Modify the TIME client so it computes E, the time that elapses between when it sends the request and when it receives a response. Add one-half E to the time the server sends.

7.5 Build a TIME client that contacts two TIME servers, and reports the differences between the times they return.

7.6 Explain how deadlock can occur if a programmer changes the line size in the TCP ECHO client to be arbitrarily large (e.g., 20,000).

7.7 The ECHO clients presented in this chapter do not verify that the text they receive back from the server matches the text they sent. Modify them to verify the data received.

7.8 The ECHO clients presented in this chapter do not count the characters sent or received. What happens if a server incorrectly sends one additional character back that the client did not send?

7.9 The example ECHO clients in this chapter do not use *shutdown.* Explain how the use of *shutdown* can improve client performance.

7.10 Rewrite the code in *UDPecho. c* so it tests reachability by generating a message, sending it, and timing the reply. If the reply does not arrive in S seconds, declare the destination host to be unreachable. Be sure to retransmit the request at least once in case the internet happens to lose a datagram.

7.11 Rewrite the code in *UDPecho. c so* it creates and sends a new message once per second, checks replies to be sure they match transmissions, and reports only the round trip time for each reply without printing the contents of the message itself.

7.12 Explain what happens to *UDPecho* when the underlying network: duplicates a request sent from the client to the server, duplicates a response sent from the server to the client, loses a request sent from the client to the server, or loses a response sent from the server to the client. Modify the code to handle each of these problems.

8
Algorithms And Issues In Server Software Design

8.1 Introduction

This chapter considers the design of server software. It discusses fundamental issues, including: connectionless vs. connection-oriented server access, stateless vs. stateful applications, and iterative vs. concurrent server implementations. It describes the advantages of each approach, and gives examples of situations in which the approach is valid. Later chapters illustrate the concepts by showing complete server programs that each implement one of the basic design ideas.

8.2 The Conceptual Server Algorithm

Conceptually, each server follows a simple algorithm: it creates a socket and binds the socket to the well-known port at which it desires to receive requests. It then enters an infinite loop in which it accepts the next request that arrives from a client, processes the request, formulates a reply, and sends the reply back to the client.

Unfortunately, this unsophisticated, conceptual algorithm suffices only for the most trivial services. To understand why, consider a service like file transfer that requires substantial time to handle each request. Suppose the first client to contact the server re quests the transfer of a giant file (e.g., 200 megabytes), while the second client to contact the server requests the transfer of a trivially small file (e.g., 20 bytes). If the server waits until the first transfer completes before starting the second transfer, the second client may wait an unreasonable amount of time for a small transfer. The second user would expect a small request to be handled immediately. Most practical servers do handle small requests quickly, because they handle more than one request at a time.

8.3 Concurrent Vs. Iterative Servers

We use the term iterative server to describe a server implementation that processes one request at a time, and the term concurrent server to describe a server that handles multiple requests at one time. Although most concurrent servers achieve apparent con currency, we will see that a concurrent implementation may not be required. it depends on the application protocol. In particular, if a server performs small amounts of processing relative to the amount of I/O it performs, it may be possible to implement the server as a single process that uses asynchronous I/O to allow simultaneous use of multiple communication channels. From a client's perspective, the server appears to communicate with multiple clients concurrently. The point is:

> *The term concurrent server refers to whether the server handles multiple requests concurrently, not to whether the underlying implementation uses multiple concurrent processes.*

In general, concurrent servers are more difficult to design and build, and the resulting code is more complex and difficult to modify. Most programmers choose concurrent server implementations, however, because iterative servers cause unnecessary delays in distributed applications and can become a performance bottleneck that affects many client applications. We can summarize:

> *Iterative server implementations, which are easier to build and understand, may result in poor performance because they make clients wait for service. In contrast, concurrent server implementations, which are more difficult to design and build, yield better performance.*

8.4 Connection-Oriented Vs. Connectionless Access

The issue of connectivity centers around the transport protocol that a client uses to access a server. In the TCP/IP protocol suite, TCP provides a *connection-oriented* transport service, while UDP provides a *connectionless* service. Thus, servers that use TCP are, by definition, *connection-oriented servers,* while those that use UDP are *connectionless servers*[1].

---

[1] The socket interface does permit an application to *connect* a UDP socket to a remote endpoint, but practical servers do not do so, and UDP is not a connection-oriented protocol.

Although we apply the terminology to servers, it would be more accurate if we restricted it to application protocols, because the choice between connectionless and connection-oriented implementations depends on the application protocol. An applica tion protocol designed to use a connection-oriented transport service may perform incorrectly or inefficiently when using a connectionless transport protocol. To summarize:

When considering the advantages and disadvantages of various server implementation strategies, the designer must remember that the application protocol used may restrict some or all of the choices.

8.5 Connection-Oriented Servers

The chief advantage of a connection-oriented approach lies in ease of programming. In particular, because the transport protocol handles packet loss and out-of-order delivery problems automatically, the server need not worry about them. Instead, a connection-oriented server manages and uses connections. It accepts an incoming connection from a client, and then sends all communication across the connection. It receives requests from the client and sends replies. Finally, the server closes the connection after it completes the interaction.

While a connection remains open, TCP provides all the needed reliability. It retransmits lost data, verifies that data arrives without transmission errors, and reorders incoming packets as necessary. When a client sends a request, TCP either delivers it reli ably or informs the client that the connection has been broken. Similarly, the server can depend on TCP to deliver responses or inform it that the connection has broken. Connection-oriented servers also have disadvantages. Connection-oriented designs require a separate socket for each connection, while connectionless designs permit communication with multiple hosts from a single socket. Socket allocation and the resulting connection management can be especially important inside an operating system that must run forever without exhausting resources. For trivial applications, the overhead of the 3-way handshake used to establish and terminate a connection makes TCP expensive compared to UDP. The most important disadvantage arises because TCP does not send any packets across an idle connection. Suppose a client establishes a connection to a server, exchanges a request and a response, and then crashes. Because the client has crashed, it will never send further requests. However, because the server has already responded to all requests received so far, it will never send more data to the client. The problem with such a situation lies in resource use: the server has data structures (including buffer space) allocated for the connection and these resources cannot be reclaimed. Remember that a server must be designed to run forever. If clients crash repeatedly, the server will run out of resources (e.g., sockets, buffer space, TCP connections) and cease to operate.

8.6 Connectionless Servers

Connectionless servers also have advantages and disadvantages. While connectionless servers do not suffer from the problem of resource depletion, they cannot depend on the underlying transport for reliable delivery. One side or the other must take responsibility for reliable delivery. Usually, clients take responsibility for retransmitting requests if no response arrives. If the server needs to divide its response into multiple data packets, it may need to implement a retransmission mechanism as well.

Achieving reliability through timeout and retransmission can be extremely difficult. In fact, it requires considerable expertise in protocol design. Because TCP/IP operates in an internet environment where end-to-end delays change quickly, using fixed values for timeout does not work. Many programmers learn this lesson the hard way when they move their applications from local area networks (which have small delays with little variation) to wider area internets (which have large delays with greater variation). To accommodate an internet environment, the retransmission strategy must be adaptive. Thus, applications must implement a retransmission scheme as complex as the one used in TCP. As a result, novice programmers are encouraged to use connection-oriented transport.

Because UDP does not supply reliable delivery, connectionless transport requires the application protocol to provide reliability, if needed, through a complex, sophisticated technique known as adaptive re-transmission. Adding adaptive retransmission to an existing application is difficult and requires considerable expertise.

Another consideration in choosing connectionless vs. connection-oriented design focuses on whether the service requires broadcast or multicast communication. Because TCP offers point-to-point communication, it cannot supply broadcast or multicast com munication; such services require UDP. Thus, any server that accepts or responds to multicast communication

must be connectionless. In practice, most sites try to avoid broadcasting whenever possible; none of the standard TCP/IP application protocols currently require multicast. However, future applications could depend more on multicast.

8.7 Failure, Reliability, And Statelessness

As Chapter 2 states, information that a server maintains about the status of ongoing interactions with clients is called state information. Servers that do not keep any state information are called stateless servers, while those that maintain state information are called stateful servers.

The issue of statelessness arises from a need to ensure reliability, especially when using connectionless transport. Remember that in an internet, messages can be duplicated, delayed, lost, or delivered out of order. If the transport protocol does not guarantee reliable delivery, and UDP does not, the application protocol must be designed to ensure it. Furthermore, the server implementation must be done carefully so it does not introduce state dependencies (and inefficiencies) unintentionally.

8.8 Optimizing Stateless Servers

To understand the subtleties involved in optimization, consider a connectionless server that allows clients to read information from files stored on the server's computer. To keep the protocol stateless, the designer requires each client request to specify a file name, a position in the file, and the number of bytes to read. The most straightforward server implementation handles each request independently: it opens the specified file, seeks to the specified position, reads the specified number of bytes, sends the information back to the client, and then closes the file.

A clever programmer assigned to write a server observes that: (1) the overhead of opening and closing files is high, (2) the clients using this server may read only á dozen bytes in each request, and (3) clients tend to read files sequentially. Furthermore, the programmer knows from experience that the server can extract data from a buffer in memory several orders of magnitude faster than it can read data from a disk. So, to optimize server performance, the programmer decides to maintain a small table of file information as Figure 8.1 shows.

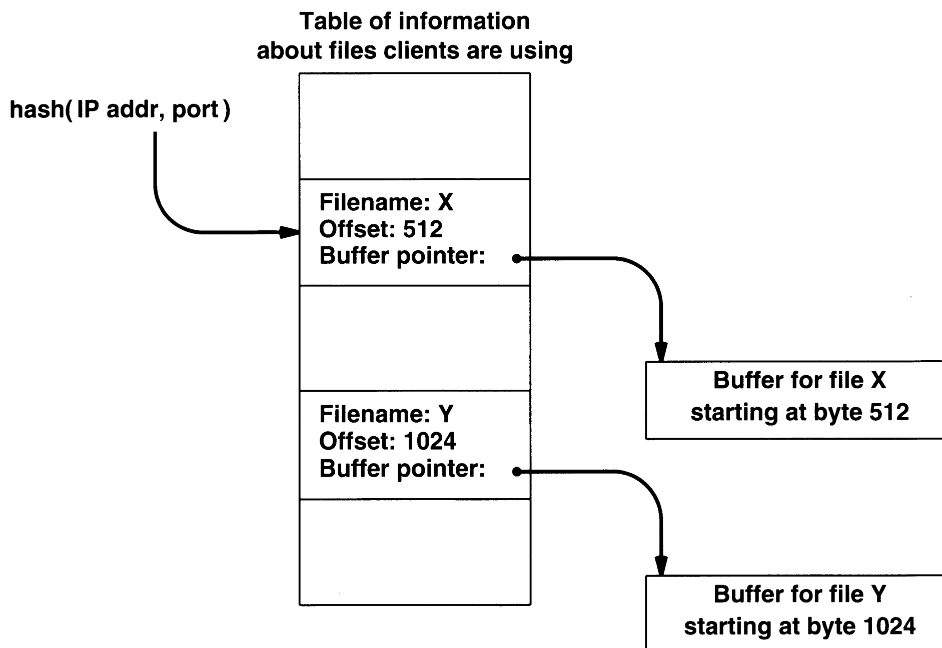

Figure 8.1 A table of information kept to improve server performance. The server uses the client's IP address and protocol port number to find an entry. This optimization introduces state information.

The programmer uses the client's IP address and protocol port number as an index into the table, and arranges for each table entry to contain a pointer to a large buffer of data from the file being read. When a client issues its first request, the server searches the table and finds that it has no record of the client. It allocates a large buffer to hold data from the file, allocates a new table entry to point to the buffer, opens the specified file, and reads data into the buffer. It then copies information out of the

buffer when forming a reply. The next time a request arrives from the same client, the server finds the matching entry in the table, follows the pointer to the buffer, and extracts data from it without opening the file. Once the client has read the entire file, the server deallocates the buffer and the table entry, making the resources available for use by another client.

Of course, our clever programmer builds the software carefully so that it checks to make sure the requested data resides in the buffer and reads new data into the buffer from the file if necessary. The server also compares the file specified in a request with the file name in the table entry to verify that the client is still using the same file as the previous request.

If the clients follow the assumptions listed above and the programmer is careful, adding large file buffers and a simple table to the server can improve its performance dramatically. Furthermore, under the assumptions given, the optimized version of the server will perform at least as fast as the original version because the server spends little time maintaining the data structures compared to the time required to read from a disk. Thus, the optimization seems to improve performance without any penalty.

Adding the proposed table changes the server in a subtle way, however, because it introduces state information. Of course, state information chosen carelessly could introduce errors in the way the server responds. For example, if the server used the client's IP address and protocol port number to find the buffer without checking the file name or file offset in the request, duplicate or out-of-order requests could cause the server to return incorrect data. But remember we said that the programmer who designed the optimized version was clever and programmed the server to check the file name and offset in each request, just in case the network duplicates or drops a request or the client decides to read from a new file instead of reading sequentially from the old file. Thus, it may seem that the addition of state information does not change the way the server replies. In fact, if the programmer is careful, the protocol will remain correct. If so, what harm can the state information do?

Unfortunately, even a small amount of state information can cause a server to perform badly when machines, client programs, or networks fail. To understand why, consider what happens if one of the client programs fails (i.e., crashes) and must be restart ed. Chances are high that the client will ask for an arbitrary protocol port number and UDP will assign a new protocol port number different from the one assigned for earlier requests. When the server receives a request from the client, it cannot know that the client has crashed and restarted, so it allocates a new buffer for the file and a new slot in the table. Consequently, it cannot know that the old table entry the client was using should be removed. If the server does not remove old entries, it will eventually run out of table slots.

It may seem that leaving an idle table entry around does not cause any problem as long as the server chooses an entry to delete when it needs a new one. For example, the server might choose to delete the least recently used (LRU) entry, much like the LRU page replacement strategy used in many virtual memory systems. However, in a network where multiple clients access a single server, frequent crashes can cause one client to dominate the table by filling it with entries that will never be reused. In the worst case, each request that arrives causes the server to delete an entry and reuse it. If one client crashes and reboots frequently enough, it can cause the server to remove entries for legitimate clients. Thus, the server expends more effort managing the table and buffers than it does answering requests[2].

The important point here is that:

> *A programmer must be extremely careful when optimizing a stateless server because managing small amounts of state information can consume resources if clients crash and reboot frequently or if the underlying network duplicates or delays messages.*

8.9 Four Basic Types Of Servers

Servers can be iterative or concurrent, and can use connection-oriented transport or connectionless transport. Figure 8.2 shows that these properties group servers into four general categories.

[2] Virtual memory systems describe this phenomenon as thrashing.

iterative connectionless	iterative connection-oriented
concurrent connectionless	concurrent connection-oriented

Figure 8.2 The four general server categories defined by whether they offer concurrency and whether they use connection-oriented transport.

8.10 Request Processing Time

In general, iterative servers suffice only for the most trivial application protocols because they make each client wait in turn. The test of whether an iterative implementation will suffice focuses on the response time needed, which can be measured locally or globally.

We define the server's request processing time to be the total time the server takes to handle a single isolated request, and we define the client's observed response time as the total delay between the time it sends a request and the time the server responds. Obviously, the response time observed by a client can never be less than the server's request processing time. However, if the server has a queue of requests to handle, the observed response time can be much greater than the request processing time.

Iterative servers handle one request at a time. If another request arrives while the server is busy handling an existing request, the system enqueues the new request. Once the server finishes processing a request, it looks at the queue to see if it has a new re quest to handle. If N denotes the average length of the request queue, the observed response time for an arriving request will be approximately N/2 + 1 times the server's request processing time. Because the observed response time increases in proportion to N, most implementations restrict N to a small value (e.g., S) and expect programmers to use concurrent servers in cases where a small queue does not suffice.

Another way of looking at the question of whether an iterative server suffices focuses on the overall load the server must handle. A server designed to handle K clients, each sending R requests per second must have a request processing time of less than *1/KR* seconds per request. If the server cannot handle requests at the required rate, its queue of waiting requests will eventually overflow. To avoid overflow in servers that may have large request processing times, a designer should consider concurrent implementations.

8.11 Iterative Server Algorithms

An iterative server is the easiest to design, program, debug, and modify. Thus, most programmers choose an iterative design whenever iterative execution provides sufficiently fast response for the expected load. Usually, iterative servers work best with simple services accessed by a connectionless access protocol. As the next sections show, however, it is possible to use iterative implementations with both connectionless and connection-oriented transport.

8.12 An Iterative, Connection-Oriented Server Algorithm

Algorithm 8.1 presents the algorithm for an iterative server accessed via the TCP connection-oriented transport. The sections following the algorithm describe each of the steps in more detail.

Algorithm 8.1

1. Create a socket and bind to the well-known address for the service being offered.
2. Place the socket in passive mode, making it ready for use by a server.
3. Accept the next connection request from the socket, and obtain a new socket for the connection.
4. Repeatedly read a request from the client, formulate a response, and send a reply back to the client according to the application protocol.
5. When finished with a particular client, close the connection and return to step *3* to accept a new connection.

Algorithm 8.1 An iterative, connection-oriented server. A single process handles connections from clients one at a time.

8.13 Binding To A Well-Known Address Using INADDR_ANY

A server needs to create a socket and bind it to the well-known port for the service it offers. Like clients, servers use procedure getportbyname to map a service name into the corresponding well-known port number. For example, TCP/IP defines an ECHO service. A server that implements ECHO uses getportbyname to map the string "echo" to the assigned port, 7.

Remember that when bind specifies a connection endpoint for a socket, it uses structure sockaddr_in, which contains both an IP address and a protocol port number. Thus, bind cannot specify a protocol port number for a socket without also specifying an IP address. Unfortunately, selecting a specific IP address at which a server will accept connections can cause difficulty. For hosts that have a single network connection, the choice is obvious because the host has only one IP address. However, routers and multi-homed hosts have multiple IP addresses. If the server specifies one particular IP address when binding a socket to a protocol port number, the socket will not accept communications that clients send to the machine's other IP addresses.

To solve the problem, the socket interface defines a special constant, INADDR_ANY, that can be used in place of an IP address. INADDR_ANY specifies a wildcard address that matches any of the host's IP addresses. Using INADDR_ANY makes it possible to have a single server on a multihomed host accept incoming communication addressed to any of the hosts's IP addresses. To summarize:

When specifying a local endpoint for a socket, servers use INADDR_ANY, instead of a specific IP address, to allow the socket to receive datagrams sent to any of the machine's IP addresses.

8.14 Placing The Socket In Passive Mode

A TCP server calls listen to place a socket in passive mode. Listen also takes an argument that specifies the length of an internal request queue for the socket. The request queue holds the set of incoming TCP connection requests from clients that have each requested a connection with the server.

8.15 Accepting Connections And Using Them

A TCP server calls accept to obtain the next incoming connection request (i.e., extract it from the request queue). The call returns the descriptor of a socket to be used for the new connection. Once it has accepted a new connection, the server uses read to obtain application protocol requests from the client, and write to send replies back. Finally, once the server finishes with the connection, it calls close to release the socket.

8.16 An Iterative, Connectionless Server Algorithm

Recall that iterative servers work best for services that have a low request processing time. Because connection-oriented transport protocols like TCP have higher overhead than connectionless transport protocols like UDP, most iterative servers use con nectionless transport. Algorithm 8.2 gives the general algorithm for an iterative server that uses UDP.

Creation of a socket for an iterative, connectionless server proceeds in the same way as for a connection-oriented server. The server's socket remains unconnected and can accept incoming datagrams from any client.

Algorithm 8.2

1. Create a socket and bind to the well-known address for the service being offered.
2. Repeatedly read the next request from a client, formulate a response, and send a reply back to the client according to the application protocol.

Algorithm 8.2 An iterative, connectionless server. A single process handles requests (datagrams) from clients one at a time.

8.17 Forming A Reply Address In A Connectionless Server

The socket interface provides two ways of specifying a remote endpoint. Chapters 6 and 7 discuss how clients use connect to specify a server's address. After a client calls connect, it can use write to send data because the internal socket data structure contains the remote endpoint address as well as the local endpoint address. A connectionless server cannot use connect, however, because doing so restricts the socket to communication with one specific remote host and port; the server cannot use the socket again to receive datagrams from arbitrary clients. Thus, a connectionless server uses an unconnected socket. It generates reply addresses explicitly, and uses the sendto socket call to specify both a datagram to be sent and an address to which it should go. Sendto has the form:

```
retcode = sendto(s , message, len, flags, toaddr, toaddrlen);
```

where s is an unconnected socket, message is the address of a buffer that contains the data to be sent, len specifies the number of bytes in the buffer, flags specifies debugging or control options, toaddr is a pointer to a sockaddr_in structure that contains the endpoint address to which the message should be sent, and toaddrlen is an integer that specifies the length of the address structure.

The socket calls provide an easy way for connectionless servers to obtain the address of a client: the server obtains the address for a reply from the source address found in the request. In fact, the socket interface provides a call that servers can use to receive the sender's address along with the next datagram that arrives. The call, recvfrom, takes two arguments that specify two buffers. The system places the arriving datagram in one buffer and the sender's address in the second buffer. A call to recvfrom has the form:

```
retcode = recvfrom(s, buf, len, flags, from, fromlen);
```

where argument s specifies a socket to use, buf specifies a buffer into which the system will place the next datagram, len specifies the space available in the buffer, from specifies a second buffer into which the system will place the source address, **and** fromlen specifies the address of an integer. Initially, fromlen specifies the length of the from buffer. When the call returns, fromlen will contain the length of the source address the system placed in the buffer. To generate a reply, the server uses the address that recvfrom stored in the from buffer when the request arrived.

8.18 Concurrent Server Algorithms

The primary reason for introducing concurrency into a server arises from a need to provide faster response times to multiple clients. Concurrency improves response time if:

- forming a response requires significant I/O,

- the processing time required varies dramatically among requests, or • the server executes on a computer with multiple processors.

In the first case, allowing the server to compute responses concurrently means that it can overlap use of the processor and peripheral devices, even if the machine has only one CPU. While the processor works to compute one response, the I/O devices can be transferring data into memory that will be needed for other responses. In the second case, timeslicing permits a single processor to handle requests that only require small amounts of processing without waiting for requests that take longer. In the third case, concurrent execution on a computer with multiple processors allows one processor to compute a response to one request while another processor computes a response to another. In fact, most concurrent servers adapt to the underlying hardware automatically - given more hardware resources (e.g., more processors), they perform better.

Concurrent servers achieve high performance by overlapping processing and I/O. They are usually designed so performance improves automatically if the server is run on hardware that offers more resources.

8.19 Master And Slave Processes

Although it is possible for a server to achieve some concurrency using a single process, most concurrent servers use multiple processes. They can be divided into two types: a single master server process begins execution initially. The master process opens a socket at the well-known port, waits for the next request, and creates a slave server process to handle each request. The master server never communicates directly with a client - it passes that responsibility to a slave. Each slave process handles communication with one client. After the slave forms a response and sends it to the client, it exits. The next sections will explain the concept of master and slave in more detail, and will show how it applies to both connectionless and connection-oriented concurrent servers.

8.20 A Concurrent, Connectionless Server Algorithm

The most straightforward version of a concurrent, connectionless server follows Algorithm 8.3.

Algorithm 8.3

Master 1. Create a socket and bind to the well-known address for the service being offered. Leave the socket unconnected.
Master 2. Repeatedly call *recvfrom* to receive the next request from a client, and create a new slave process to handle the response.
Slave 1. Receive a specific request upon creation as well as access to the socket.
Slave 2. Form a reply according to the application protocol and send it back to the client using *sendto*.
Slave 3. Exit (i.e., a slave process terminates after handling one request).

Algorithm 8.3 A concurrent, connectionless server. The master server process accepts incoming requests (datagrams) and creates a slave process to handle each.

Programmers should remember that although the exact cost of creating a process depends on the operating system and underlying architecture, the operation can be expensive. In the case of a connectionless protocol, one must consider carefully whether the cost of concurrency will be greater than the gain in speed. In fact:

Because process creation is expensive, few connectionless servers have concurrent implementations.

8.21 A Concurrent, Connection-Oriented Server Algorithm

Connection-oriented application protocols use a connection as the basic paradigm for communication. They allow a client to establish a connection to a server, communicate over that connection, and then discard it. In most cases, the connection between client and server handles more than a single request: the protocol allows a client to repeatedly send requests and receive responses without terminating the connection or creating a new one. Thus, Connection-oriented servers implement concurrency among connections rather than among individual requests.

Algorithm 8.4 specifies the steps that a concurrent server uses for a connectionoriented protocol.

Algorithm 8.4

Master 1. Create a socket and bind to the well-known address for the service being offered. Leave the socket unconnected.
Master 2. Place the socket in passive mode, making it ready for use by a server.
Master 3. Repeatedly call *accept* to receive the next request from a client, and create a new slave process to handle the response.
Slave 1. Receive a connection request (i.e., socket for the connection) upon creation.
Slave 2. Interact with the client using the connection: read request(s) and send back response(s).
Slave 3. Close the connection and exit. The slave process exits after handling all requests from one client.

Algorithm 8.4 A concurrent, connection-oriented server. The master server process accepts incoming connections and creates a slave process to handle each. Once the slave finishes, it closes the connection.

As in the connectionless case, the master server process never communicates with the client directly. As soon as a new connection arrives, the master creates a slave to handle that connection. While the slave interacts with the client, the master waits for other connections.

8.22 Using Separate Programs As Slaves

Algorithm 8.4 shows how a concurrent server creates a new process for each connection. In UNIX, the master server does so by calling the *fork* system call. For simple application protocols, a single server program can contain all the code needed for both the master and slave processes. After the call to *fork*, the original process loops back to accept the next incoming connection, while the new process becomes the slave and handles the connection. In some cases, however, it may be more convenient to have the slave process execute code from a program that has been written and compiled independently. UNIX can handle such cases easily because it allows the slave process to call execve after the call to fork. The general idea is:

For many services, a single program can contain code for both the master and server processes. In cases where an independent program makes the slave process easier to program or understand, the master program contains a call to execve after the call to fork.

8.23 Apparent Concurrency Using A Single Process

Previous sections discuss concurrent servers implemented with concurrent processes. In some cases, however, it makes sense to use a single process to handle client requests concurrently. In particular, some operating systems make process crea tion so expensive that a server cannot afford to create a new process for each request or each connection. More important, many applications require the server to share information among all connections.

To understand the motivation for a server that provides *apparent concurrency* with a single process, consider the X window system. X allows multiple clients to paint text and graphics in windows that appear on a bit-mapped display. Each client

controls one window, sending requests that update the contents. Each client operates independently, and may wait many hours before changing the display or may update the display frequently. For example, an application that displays the time by drawing a picture of a clock might update its display every minute. Meanwhile, an application that displays the status of a user's electronic mail waits until new mail arrives before it changes the display.

A server for the X window system integrates information it obtains from clients into a single, contiguous section of memory called the *display buffer*. Because data arriving from all clients contributes to a single, shared data structure and because BSD UNIX does not allow independent processes to share memory, the server cannot execute as separate UNIX processes. Thus, a conflict arises between a desire for concurrency among processes that share memory and a lack of support for such concurrency in UNIX.

Although it may not be possible to achieve *real concurrency* among processes that share memory, it may be possible to achieve *apparent concurrency* if the total load of requests presented to the server does not exceed its capacity to handle them. To do so, the server operates as a single UNIX process that uses the *select* system call for asynchronous I/O. Algorithm 8.5 describes the steps a single-process server takes to handle multiple connections.

Algorithm 8.5

1. Create a socket and bind to the well-known port for the service. Add socket to the list of those on which I/O is possible.
2. Use *select* to wait for I/O on existing sockets.
3. If original socket is ready, use *accept* to obtain the next connection, and add the new socket to the list of those on which I/O is possible.
4. If some socket other than the original is ready, use *read* to obtain the next request, form a response, and use *write* to send the response back to the client.
5. Continue processing with step 2 above.

Algorithm 8.5 A concurrent, connection-oriented server implemented by a single process. The server process waits for the next descriptor that is ready, which could mean a new connection has arrived or that a client has sent a request on an existing connection.

8.24 When To Use Each Server Type

Iterative vs. Concurrent: Iterative servers are easier to design, implement, and maintain, but concurrent servers can provide quicker response to requests. Use an iterative implementation if request processing time is short and an iterative solution produces response times that are sufficiently fast for the application.

Real vs. Apparent Concurrency: A single-process server must manage multiple connections and use asynchronous I/O; a multi-process server allows the operating system to provide concurrency automatically. Use a single-process solution if the server must share or exchange data among connections. Use a multi-process solution if each slave can operate in isolation or to achieve maximal concurrency (e.g., on a multiprocessor).

Connection-Oriented vs. Connectionless: Because connection-oriented access means using TCP, it implies reliable delivery. Because connectionless transport means using UDP, it implies unreliable delivery. Only use connectionless transport if the ap plication protocol handles reliability (almost none do) or each client accesses its server on a local area network that exhibits extremely low loss and no packet reordering. Use connection-oriented transport whenever a wide area network separates the client and server. Never move a connectionless client and server to a wide area environment without checking to see if the application protocol handles the reliability problems.

8.25 A Summary of Server Types Iterative, Connectionless Server

The most common form of connectionless server, used especially for services that require a trivial amount of processing for each request. Iterative servers are often stateless, making them easier to understand and less susceptible to failures.

Iterative, Connection-Oriented Server

A less common server type used for services that require a trivial amount of processing for each request, but for which reliable transport is necessary. Because the overhead associated with establishing and terminating connections can be high, the average response time can be non-trivial.

Concurrent, Connectionless Server

An uncommon type in which the server creates a new process to handle each request. On many systems, the added cost of process creation dominates the added efficiency gained from concurrency. To justify concurrency, either the time required to create a new process must be significantly less than the time required to compute a response or concurrent requests must be able to use many I/O devices simultaneously.

Concurrent, Connection-Oriented Server

The most general type of server because it offers reliable transport (i.e., it can be used across a wide area internet) as well as the ability to handle multiple requests concurrently. Two basic implementations exist: the most common implementation uses concurrent processes to handle connections; a far less common implementation relies on a single process and asynchronous I/O to handle multiple connections.

In a concurrent process implementation, the master server process creates a slave process to handle each connection. Using multiple processes makes it easy to execute a separately compiled program for each connection instead of writing all the code in a single, large server program.

In the single-process implementation, the server process manages multiple connections. It achieves apparent concurrency by using asynchronous I/O. The process repeatedly waits for I/O on any of the connections it has open and handles that request. Because a single process handles all connections, it can share data among them. However, because the server has only one process, it cannot handle requests faster than an iterative server, even on a computer that has multiple processors. The application must need data sharing or the processing time for each request must be short to justify this server implementation.

8.26 The Important Problem Of Server Deadlock

Many server implementations share an important flaw: namely, the server can be subject to deadlock[3]. To understand how deadlock can happen, consider an iterative, connection-oriented server. Suppose some client application, *C*, misbehaves. In the simplest case, assume C makes a connection to a server, but never sends a request. The server will accept the new connection, and call read to extract the next request. The server process blocks in the call to *read* waiting for a request that will never arrive.

Server deadlock can arise in a much more subtle way if clients misbehave by not consuming responses. For example, assume that a client *C* makes a connection to a server, sends it a sequence of requests, but never reads the responses. The server keeps accepting requests, generating responses, and sending them back to the client. At the server, TCP protocol software transmits the first few bytes over the connection to the client. Eventually, TCP will fill the client's receive window and will stop transmitting data. If the server application program continues to generate responses, the local buffer TCP uses to store outgoing data for the connection will become full and the server process will block.

Deadlock arises because processes block when the operating system cannot satisfy a system call. In particular, a call to *write* will block the calling process if TCP has no local buffer space for the data being sent; a call to *read* will block the calling process until TCP receives data. For concurrent servers, only the single slave process associated with a particular client blocks if the client fails to send requests or read responses. For a single-process implementation, however, the central server process will

[3] The term *deadlock* refers to a condition in which a program or set of programs cannot proceed because they are blocked waiting for an event that will never happen. In the case of servers, deadlock means that the server ceases to answer requests.

block. If the central server process blocks, it cannot handle other connections. The important point is that any server using only one process can be subject to deadlock.

> *A misbehaving client can cause deadlock in a single-process server if the server uses system functions that can block when communicating with the client. Deadlock is a serious liability in servers because it means the behavior of one client can prevent the server from handling other clients.*

8.27 Alternative Implementations

Chapters 9 through 12 provide examples of the server algorithms described in this chapter. Chapters 13 and 14 extend the ideas by discussing two important practical implementation techniques not described here: multiprotocol and multiservice servers. While both techniques provide interesting advantages for some applications, they have not been included here because they are best understood as simple generalizations of the single-process server algorithm illustrated in Chapter 12.

8.28 Summary

Conceptually, a server consists of a simple algorithm that iterates forever, waiting for the next request from a client, handling the request, and sending a reply. In practice, however, servers use a variety of implementations to achieve reliability, flexibility, and efficiency.

Iterative implementations work well for services that require little computation. When using a connection-oriented transport, an iterative server handles one connection at a time; for connectionless transport, an iterative server handles one request at a time.

To achieve efficiency, servers often provide concurrent service by handling multiple requests at the same time. A connection-oriented server provides for concurrency among connections by creating a process to handle each new connection. A connec tionless server provides concurrency by creating a new process to handle each new re**quest.**

Any server implemented with a single process that uses synchronous system functions like *read* or *write* can be subject to deadlock. Deadlock can arise in iterative servers as well as in concurrent servers that use a single-process implementation. Server deadlock is especially serious because it means a single misbehaving client can prevent the server from handling requests for other clients.

FOR FURTHER STUDY

Stevens [1990] describes some of the server algorithms covered in this chapter and shows implementation details. BSD UNIX contains examples of many server algorithms; programmers often consult the UNIX source code for programming techniques.

EXERCISES

8.1 Calculate how long an iterative server takes to transfer a 200 megabyte file if the internet has a throughput of 2.3 Kbytes per second.

8.2 If 20 clients each send 2 requests per second to an iterative server, what is the maximum time that the server can spend on each request?

8.3 How long does it take a concurrent, connection-oriented server to accept a new connection and create a new process to handle it on the computers to which you have access?

8.4 Write an algorithm for a concurrent, connectionless server that creates one new process for each request.

8.5 Modify the algorithm in the previous problem so the server creates one new process per client instead of one new process per request. How does your algorithm handle process termination?

8.6 Connection-oriented servers provide concurrency among connections. Does it make sense for a concurrent, connection-oriented server to increase concurrency even further by having the slave processes create additional processes for each request? Explain.

8.7 Rewrite the TCP echo client so it uses a single process to concurrently handle input from the keyboard, input from its TCP connection, and output to its TCP connection.

8.8 Can clients cause deadlock or disrupt service in concurrent servers? Why or why not?

8.9 Look carefully at the select system call. How can a single-process server use select to avoid deadlock?

8.10 The *select* call takes an argument that specifies how many I/O descriptors it should check. Explain how the argument makes a single-process server program portable across many UNIX systems.

9
Iterative, Connectionless Servers (UDP)

9.1 Introduction

The previous chapter discusses many possible server designs, comparing the advantages and disadvantages of each. This chapter gives an example of an iterative server implementation that uses connectionless transport. The example server follows Algorithm 8.2[1]. Later chapters continue the discussion by providing example implementations of other server algorithms.

9.2 Creating A Passive Socket

The steps required to create a passive socket are similar to those required to create an active socket. They involve many details, and require the program to look up á serice name to obtain a well-known protocol port number.

To help simplify server code, programmers should use procedures to hide the details of socket allocation. As in the client examples, our example implementations use two high-level procedures, *passiveUDP* and *passiveTCP,* that allocate a passive socket and bind it to the server's well-known port. Each server invokes one of these procedures, with the choice dependent on whether the server uses connectionless or connection-oriented transport. This chapter considers *passive UDP;* the next chapter show s the code for *passiveTCP.* Because the two procedures have many details in common. they both call the low-level procedure, *passivesock* to perform the work.

A connectionless server calls function *passiveUDP* to create a socket for the service that it offers. If the server needs to use one of the ports reserved for well-known services (i.e., a port numbered below 1024), the server process must have special privilege[2]. An arbitrary application program can use *passiveUDP* to create a socket for nonprivileged services. *PassiveUDP* calls *passivesock* to create a connectionless socket, and then returns the socket descriptor to its caller.

To make it easy to test client and server software, *passivesock* relocates all port values by adding the contents of global integer *portbase.* The importance of using *portbase* will become clearer in later chapters. However, the basic idea is fairly easy to understand:

> *If a new version of a client-server application uses the same protocol port numbers as an existing, production version, the new software cannot be tested while the production version continues to execute.*

Using *portbase* allows a programmer to compile a modified version of a server, and then to have the server look up the standard protocol port and compute a final port number as a function of the standard port and the value of *portbase. If* the programmer selects a unique value *of portbase* for each particular version of a client-server pair, the ports used by the new version will not conflict with the ports used by the production version. In fact, using *portbase* makes it possible to test multiple versions of a clientserver pair at the same time without interference because each pair communicates independently of other pairs.

```
/* passiveUDP.c - passiveUDP */

int passivesock(const char *service, const char *transport,
        int qlen);

/*------------------------------------------------------------------------
 * passiveUDP - create a passive socket for use in a UDP server
```

---

[1] See page Section 8.16 for a description of Algorithm 8.2.

[2] In UNIX, an application must execute as root (i.e., be the superuser) to have sufficient privilege to bind to a reserved port.

```
 *------------------------------------------------------------------------
 */
int
passiveUDP(const char *service)
/*
 * Arguments:
 *      service - service associated with the desired port
 */
{
    return passivesock(service, "udp", 0);
}
```

Procedure *passivesock* contains the socket allocation details, including the use of *portbase*. It takes three arguments. The first argument specifies the name of a service, the second specifies the name of the protocol, and the third (used only for TCP sockets) specifies the desired length of the connection request queue. *Passivesock* allocates either a datagram or stream socket, binds the socket to the well-known port for the service, and returns the socket descriptor to its caller.

Recall that when a server binds a socket to a well-known port, it must specify the address using structure *sockaddr_in,* which includes an IP address as well as a protocol port number. *Passivesock* uses the constant *INADDR_ANY* instead of a specific local IP address, enabling it to work either on hosts that have a single IP address or on routers and multi-homed hosts that have multiple IP addresses. Using *INADDR_ANY* means that the server will receive communication addressed to its well-known port at any of the machine's IP addresses.

```
/* passivesock.c - passivesock */

#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>

#include <stdlib.h>
#include <string.h>
#include <netdb.h>

extern int  errno;
typedef unsigned short u_short;
int errexit(const char *format, ...);

u_short portbase = 0;       /* port base, for non-root servers  */

/*------------------------------------------------------------------------
 * passivesock - allocate & bind a server socket using TCP or UDP
 *------------------------------------------------------------------------
 */
```

```
int
passivesock(const char *service, const char *transport, int qlen)
/*
 * Arguments:
 *      service   - service associated with the desired port
 *      transport - transport protocol to use ("tcp" or "udp")
 *      qlen      - maximum server request queue length
 */
{
    struct servent  *pse;   /* pointer to service information entry */
    struct protoent *ppe;   /* pointer to protocol information entry*/
    struct sockaddr_in sin; /* an Internet endpoint address     */
    int s, type;    /* socket descriptor and socket type    */

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;

    /* Map service name to port number */
    if ( pse = getservbyname(service, transport) )
        sin.sin_port = htons(ntohs((u_short)pse->s_port)
            + portbase);
    else if ( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
        errexit("can't get \"%s\" service entry\n", service);

    /* Map protocol name to protocol number */
    if ( (ppe = getprotobyname(transport)) == 0)
        errexit("can't get \"%s\" protocol entry\n", transport);

    /* Use protocol to choose a socket type */
    if (strcmp(transport, "udp") == 0)
        type = SOCK_DGRAM;
    else
        type = SOCK_STREAM;

    /* Allocate a socket */
    s = socket(PF_INET, type, ppe->p_proto);
    if (s < 0)
        errexit("can't create socket: %s\n", strerror(errno));

    /* Bind the socket */
```

```
    if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't bind to %s port: %s\n", service,
            strerror(errno));
    if (type == SOCK_STREAM && listen(s, qlen) < 0)
        errexit("can't listen on %s port: %s\n", service,
            strerror(errno));
    return s;
}
```

9.3 Process Structure

Figure 9.1 illustrates the simple process structure used for an iterative, connectionless server.

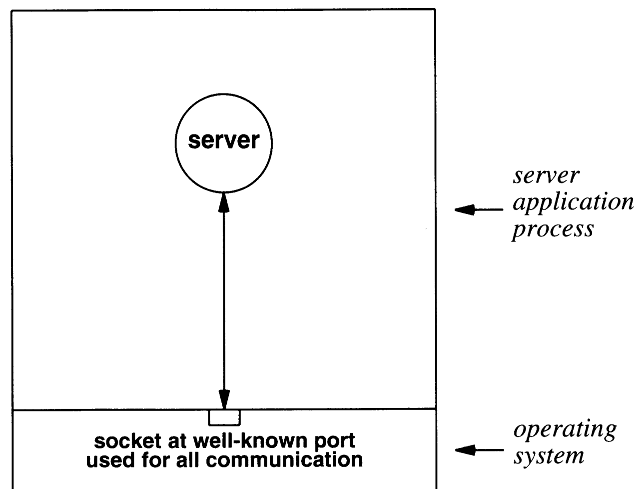

Figure 9.1 The process structure for an iterative, connectionless server. A single server process communicates with many clients using one socket.

The single server process executes forever. It uses a single passive socket that has been bound to the well-known protocol port for the service it offers. The server obtains a request from the socket, computes a response, and sends a reply back to the client using the same socket. The server uses the source address in the request as the destination address in the reply.

9.4 An Example TIME Server

An example will illustrate how a connectionless server process uses the socket allocation procedures described above. Recall from Chapter 7 that clients use the TIME service to obtain the current time of day from a server on another system. Because TIME requires little computation, an iterative server implementation works well. File *UDPtimed.c* contains the code for an iterative, connectionless TIME server.

```
/* UDPtimed.c - main */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <stdio.h>
```

```
#include <time.h>
#include <string.h>

typedef unsigned long u_long;

extern int  errno;

int passiveUDP(const char *service);
int errexit(const char *format, ...);

#define UNIXEPOCH   2208988800  /* UNIX epoch, in UCT secs  */

/*------------------------------------------------------------------------
 * main - Iterative UDP server for TIME service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    struct sockaddr_in fsin;    /* the from address of a client */
    char    *service = "time";  /* service name or port number  */
    char    buf[1];         /* "input" buffer; any size > 0 */
    int sock;           /* server socket        */
    time_t  now;            /* current time         */
    int alen;           /* from-address length      */

    switch (argc) {
    case    1:
        break;
    case    2:
        service = argv[1];
        break;
    default:
        errexit("usage: UDPtimed [port]\n");
    }

    sock = passiveUDP(service);

    while (1) {
        alen = sizeof(fsin);
        if (recvfrom(sock, buf, sizeof(buf), 0,
```

```
                (struct sockaddr *)&fsin, &alen) < 0)
            errexit("recvfrom: %s\n", strerror(errno));
        (void) time(&now);
        now = htonl((u_long)(now + UNIXEPOCH));
        (void) sendto(sock, (char *)&now, sizeof(now), 0,
            (struct sockaddr *)&fsin, sizeof(fsin));
    }
}
```

Like any server, the *UDPtimed* process must execute forever. Thus, the main body of code consists of an infinite loop that accepts a request, computes the current time, and sends a reply back to the client that sent the request.

The code contains several details. After parsing its arguments, *UDPtimed* calls *passive UDP* to create a passive socket for the TIME service. It then enters the infinite loop. The TIME protocol specifies that a client can send an arbitrary datagram to trigger a reply. The datagram can be of any length and can contain any values because the server does not interpret its contents. The example implementation uses *recvfrom* to read the next datagram. *Recvfrom* places the incoming datagram in buffer *buf,* and places the endpoint address of the client that sent the datagram in structure *fsin.* Because it does not need to examine the data, the implementation uses a single-character buffer. If the datagram contains more than one byte of data, *recvfrom* discards all remaining bytes.

UDPtimed uses the UNIX system routine *time* to obtain the current time. Recall from Chapter 7 that UNIX uses a 32-bit integer to represent time, measuring from the epoch of midnight, January 1, 1970. After obtaining the time from UNIX, *UDPtimed* must convert it to a value measured from the Internet epoch and place the result in network byte order. To perform the conversion, it adds constant *UNIXEPOCH,* which is defined to have the value 2208988800, the difference in seconds between the Internet epoch and the UNIX epoch. It then calls function *htonl* to convert the result to network byte order. Finally, *UDPtimed* calls *sendto* to transmit the result back to the client. *Sendto* uses the endpoint address in structure *fsin* as the destination address (i.e., it uses the address of the client that sent the datagram).

9.5 Summary

For simple services, where a server does little computation for each request, an iterative implementation works well. This chapter presented an example of an iterative server for the TIME service that uses UDP for connectionless access. The example il lustrates how procedures hide the details of socket allocation and make the server code simpler and easier to understand.

FOR FURTHER STUDY

Harrenstien [RFC 738] specifies the TIME protocol. Mills [RFC 1305] describes the Network Time Protocol (NTP); Mills [September 1991] summarizes issues related to using NTP in practical networks, and Mills [RFC 1361] discusses the use of NTP for clock synchronization. Marzullo and Owicki [July 1985] also discusses how to maintain clocks in a distributed environment.

EXERCISES

9.1 Instrument *UDPtimed* to determine how much time it expends processing each request. If you have access to a network analyzer, also measure the time that elapses between the request and response packets.

9.2 Suppose *UDPtimed* inadvertently clobbered the client's address between the time it received a request and sent a response (i.e., the server accidentally assigned *fsin* a random value before using it in the call to *sendto).* What would happen? Why?

9.3 Conduct an experiment to determine what happens if *N* clients all send requests to *UDPtimed* simultaneously. Vary both *N,* the number of senders, and S, the size of the datagrams they send. Explain why the server fails to respond to all requests. (Hint: look at the manual page for *listen.)*

9.4 The example code in *UDPtimed. c* specifies a buffer size of 1 when it calls *recvfrom.* What happens if it specifies a buffer size of 0?

9.5 Compute the difference between the UNIX time epoch and the Internet time epoch. Remember to account for leap years. Does the value you compute agree with the constant *UNIXEPOCH* defined in *UDPtimed?* If not, explain. (Hint: read about leap seconds.)

9.6 As a security check, the system manager asks you to modify *UDPtimed* so it keeps a written log of all clients who access the service. Modify the code to print a line on the console whenever a request arrives. Explain how logging can affect the service.

9.7 If you have access to a pair of machines connected by a wide-area internet, use the *UDPtime* client in Chapter 7 and the *UDPtimed* server in this chapter to see if your internet drops or duplicates packets.

10
Iterative, Connection-Oriented Servers (TCP)

10.1 Introduction

The previous chapter provides an example of an iterative server that uses UDP for connectionless transport. This chapter shows how an iterative server can use TCP for connection-oriented transport. The example server follows Algorithm 8.1[1].

10.2 Allocating A Passive TCP Socket

Chapter 9 mentions that a connection-oriented server uses function *passiveTCP* to allocate a stream socket and bind it to the well-known port for the service being offered. *PassiveTCP* takes two arguments. The first argument, a character string, specifies the name or number of a service, and the second specifies the desired length of the incoming connection request queue. If the first argument contains a name, it must match one of the entries in the service database accessed by library function *getservbyname*. If the first argument specifies a port number, it must represent the number as a text string (e. g., "79").

```
/* passiveTCP.c - passiveTCP */


int passivesock(const char *service, const char *transport,
        int qlen);


/*------------------------------------------------------------------------
 * passiveTCP - create a passive socket for use in a TCP server
 *------------------------------------------------------------------------
 */
int
passiveTCP(const char *service, int qlen)
/*
 * Arguments:
 *      service - service associated with the desired port
 *      qlen    - maximum server request queue length
 */
{
    return passivesock(service, "tcp", qlen);
}
```

10.3 A Server For The DAYTIME Service

Recall from Chapter 7 that the DAYTIME service allows a user on one machine to obtain the current date and time of day from another machine. Because the DAYTIME service is intended for humans, it specifies that the server must format the date in an easily readable string of ASCII text when it sends a reply, Thus, the client can display the response for a user exactly as it is received.

---

[1] See Section 8.12 for a description of Algorithm 8.1.

Chapter 7 shows how a client uses TCP to contact a DAYTIME server and to display the text that the server sends back. Because obtaining and formatting a date requires little processing and one expects little demand for the service, a DAYTIME server need not be optimized for speed. If additional clients attempt to make connection requests while the server is busy handling a request, the protocol software enqueues the additional requests. Thus, an iterative implementation suffices.

10.4 Process Structure

As Figure 10.1 shows, an iterative, connection-oriented server uses a single process. The process iterates forever, using one socket to handle incoming requests and a second, temporary socket to handle communication with a client.

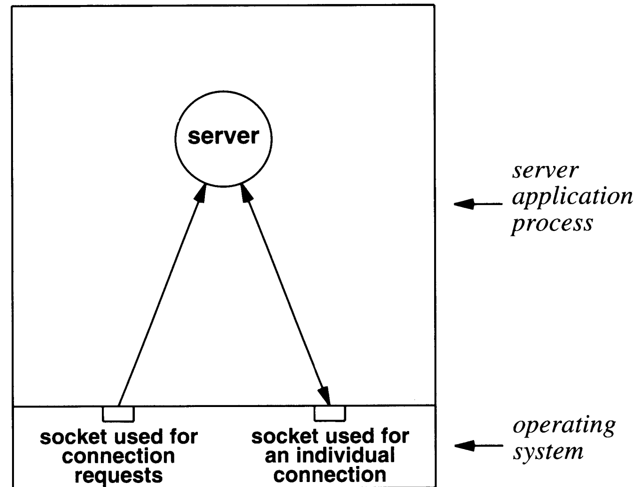

Figure 10.1 The process structure of an iterative, connection-oriented server. The server waits at the well-known port for a connection, and then communicates with the client over that connection.

A server that uses connection-oriented transport iterates on connections: it waits at the well-known port for the next connection to arrive from a client, accepts the connection, handles it, closes the connection, and then waits again. The DAYTIME service makes the implementation especially simple because the server does not need to receive an explicit request from the client - it uses the presence of an incoming connection to trigger a response. Because the client does not send an explicit request, the server does not read data from the connection.

10.5 An Example DAYTIME Server

File *TCPdaytimed. c* contains example code for an iterative, connection-oriented DAYTIME server.

```
/* TCPdaytimed.c - main */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <unistd.h>
#include <stdio.h>
#include <string.h>

extern int  errno;
int     errexit(const char *format, ...);
```

```
int     TCPdaytimed(int fd);
int     passiveTCP(const char *service, int qlen);

#define QLEN    5

/*------------------------------------------------------------------------
 * main - Iterative TCP server for DAYTIME service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    struct  sockaddr_in fsin;   /* the from address of a client */
    char    *service = "daytime";   /* service name or port number  */
    int msock, ssock;       /* master & slave sockets   */
    int alen;           /* from-address length      */

    switch (argc) {
    case    1:
        break;
    case    2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPdaytimed [port]\n");
    }

    msock = passiveTCP(service, QLEN);

    while (1) {
        ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
        if (ssock < 0)
            errexit("accept failed: %s\n", strerror(errno));
        (void) TCPdaytimed(ssock);
        (void) close(ssock);
    }
}

/*------------------------------------------------------------------------
 * TCPdaytimed - do TCP DAYTIME protocol
 *------------------------------------------------------------------------
```

```
 */
int
TCPdaytimed(int fd)
{
    char    *pts;           /* pointer to time string   */
    time_t  now;            /* current time         */
    char    *ctime();

    (void) time(&now);
    pts = ctime(&now);
    (void) write(fd, pts, strlen(pts));
    return 0;
}
```

Like the iterative, connectionless server described in the previous chapter, an iterative, connection-oriented server must run forever. After creating a socket that listens at the well-known port, the server enters an infinite loop in which it accepts and handles connections.

The code for the server is fairly short because the call to *passiveTCP* hides the details of socket allocation and binding. The call to *passiveTCP* creates a master socket associated with the well-known port for the DAYTIME service. The second argument specifies that the master socket will have a request queue length *of QLEN,* allowing the system to enqueue connection requests that arrive from *QLEN* additional clients while the server is busy replying to a request from a given client.

After creating the master socket, the server's main program enters an infinite loop. During each iteration of the loop, the server calls *accept* to obtain the next connection request from the master socket. To prevent the server from consuming resources while waiting for a connection from a client, the call to *accept* blocks the server process until a connection arrives. When a connection request arrives, the TCP protocol software engages in a 3-way handshake to establish a connection. Once the handshake completes and the system allocates a new socket for the incoming connection, the call to *accept* returns the descriptor of the new socket, allowing the server to continue execution. If no connection arrives, the server process remains blocked forever in the *accept* call.

Each time a new connection arrives, the server calls procedure *TCPdaytimed* to handle it. The code in *TCPdaytimed* centers around calls to the UNIX functions *time* and *ctime*. Procedure *time* returns a 32-bit integer that gives the current time in seconds since the UNIX epoch. The UNIX library function *ctime* takes an integer argument that specifies a time in seconds since the UNIX epoch, and returns the address of an ASCII string that contains the time and date formatted so a human can understand it. Once the server obtains the time and date in an ASCII string, it calls *write* to send the string back to the client over the TCP connection.

Once the call to *TCPdaytimed* returns, the main program continues executing the loop, and encounters the *accept* call again. The *accept* call blocks the server until another request arrives.

10.6 Closing Connections

After it has written the response, the call to procedure *TCPdaytimed* returns. Once the call returns, the main program explicitly closes the socket on which the connection arrived.

Calling *close* requests a graceful shutdown. In particular, TCP guarantees that all data will be reliably delivered to the client and acknowledged before it terminates the connection. Thus, when calling *close*, a programmer does not need to worry about data being delivered.

Of course, TCP's definition of graceful shutdown means that the call to *close* may not return instantly - the call will block until TCP on the server receives a reply from TCP on the client. Once the client acknowledges both the receipt of all data and the request to terminate the connection, the *close* call returns,

10.7 Connection Termination And Server Vulnerability

The application protocol determines how a server manages TCP connections. In particular, the application protocol usually dictates the choice of the termination strategy. For example, arranging for the server to close connections works well for the DAYTIME protocol because the server knows when it has finished sending data. Applications that have more complex client-server interactions cannot choose to have the server close a connection immediately after processing one request because they must wait to see if the client chooses to send additional request messages. For example, consider an ECHO server. The client controls server processing because it determines the amount of data to be echoed. Because the server must process arbitrary amounts of data, it cannot close the connection after reading and writing once. Thus, the client must signal completion so the server knows when to terminate the connection.

Allowing a client to control connection duration can be dangerous because it allows clients to control resource use. In particular, misbehaving clients can cause the server to consume resources like sockets and TCP connections. It may seem that our example server will never run out of resources because it explicitly closes connections. Even our simple connection termination strategy can be vulnerable to misbehaving clients. To understand why, recall that TCP defines a connection timeout period of 2 times the maximum segment lifetime (2*MSL) after a connection closes. During the timeout, TCP keeps a record of the connection so it can correctly reject any old packets that may have been delayed. Thus, if clients make repeated requests rapidly, they can use up resources at the server. Although programmers may have little control over protocols, they should understand how protocols can make distributed software vulnerable to network failures and try to avoid such vulnerabilities when designing servers.

10.8 Summary

An iterative, connection-oriented server iterates once per connection. Until a connection request arrives from a client, the server remains blocked in a call to *accept.* Once the underlying protocol software establishes the new connection and creates a new socket, the call to *accept* returns the socket descriptor and allows the server to continue execution.

Recall from Chapter 7 that the DAYTIME protocol uses the presence of a connection to trigger a response from the server. The client does not need to send a request because the server responds as soon as it detects a new connection. To form a response, the server obtains the current time from the operating system, formats the information into a string suitable for humans to read, and then sends the response back to the client. The example server closes the socket that corresponds to an individual connection after sending a response. The strategy of closing the connection immediately works because the DAYTIME service only allows one response per connection. Servers that allow multiple requests to arrive over a single connection must wait for the client to close the connection.

FOR FURTHER STUDY

Postel [RFC 867] describes the DAYTIME protocol used in this chapter.

EXERCISES

10.1 Does a process need special privilege to run a DAYTIME server on your local system? Does it need special privilege to run a DAYTIME client?

10.2 What is the chief advantage of using the presence of a connection to trigger a response from a server? The chief disadvantage?

10.3 Some DAYTIME servers terminate the line of text by a combination of two characters: *carriage return (CR)* and *linefeed (LF).* Modify the example server to send *CR-LF* at the end of the line instead of sending only *LF.* How does the standard specify lines should be terminated?

10.4 TCP software usually allocates a fixed-size queue for additional connection requests that arrive while a server is busy, and allows the server to change the queue size using *listen.* How large is the queue that your local TCP software provides? How large can the server make the queue with *listen?*

10.5 Modify the example server code in *TCPdaytimed. c* so it does not explicitly close the connection after writing a response. Does it still work correctly? Why or why not?

10.6 Compare a connection-oriented server that explicitly closes each connection after sending a response to one that allows the client to hold a connection arbitrarily long before closing the connection. What are the advantages and disadvantages of each approach?

10.7 Assume that TCP uses a connection timeout of 4 minutes (i.e., keeps information for 4 minutes after a connection closes). If a DAYTIME server runs on a system that has 100 slots for TCP connection information, what is the maximum rate at which the server can handle requests without running out of slots?

11
Concurrent, Connection-Oriented Servers (TCP)

11.1 Introduction

The previous chapter illustrates how an iterative server uses a connection-oriented transport protocol. This chapter gives an example of a concurrent server that uses a connection-oriented transport. The example server follows Algorithm 8.4[1], the design that programmers use most often when they build concurrent TCP servers. The server relies on the operating system's support for concurrent processing to achieve concurrency when computing responses. The system manager arranges to have the master server process start automatically when the system boots. The master server runs forever waiting for new connection requests to arrive from clients. The master creates a new slave process to handle each new connection, and allows each slave to handle all communication with its client.

Later chapters consider alternative implementations of concurrent servers, and show how to extend the basic ideas presented here.

11.2 Concurrent ECHO

Consider the ECHO service described in Chapter 7. A client opens a connection to a server, and then repeatedly sends data across the connection and reads the "echo" the server returns. The ECHO server responds to each client. It accepts a connection, reads from the connection, and then sends back the same data it receives.

To allow a client to send arbitrary amounts of data, the server does not read the entire input before it sends a response. Instead, it alternates between reading and writing. When a new connection arrives, the server enters a loop. On each iteration of the loop, the server first reads from the connection and then writes the data back to the connection. The server continues iterating until it encounters an end-of-file condition, at which time it closes the connection.

11.3 Iterative Vs. Concurrent Implementations

An iterative implementation of an ECHO server can perform poorly because it requires a given client to wait while it handles all prior connection requests. If a client chooses to send large amounts of data (e.g., many megabytes), an iterative server will delay all other clients until it can satisfy the request.

A concurrent implementation of an ECHO server avoids long delays because it does not allow a single client to hold all resources. Instead, a concurrent server allows its communication with many clients to proceed simultaneously. Thus, from a client's point of view, a concurrent server offers better observed response time than an iterative server.

11.4 Process Structure

Figure 11.1 illustrates the process structure of a concurrent, connection-oriented server. As the figure shows, the master server process does not communicate with clients directly. Instead, it merely waits at the well-known port for the next connection request. Once a request has arrived, the system returns the socket descriptor of the new socket to use for that connection. The master server process creates a slave process to handle the connection, and allows the slave to operate concurrently. At any time, the server consists of one master process and zero or more slave processes.

---

[1] See Section 8.21 for a description of Algorithm 8.4.

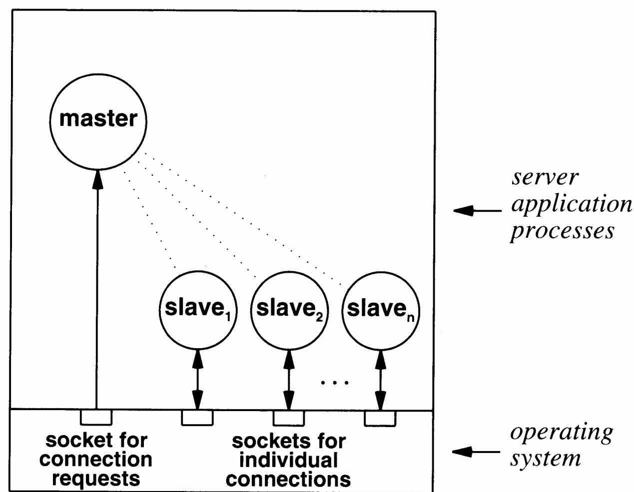

Figure 11.1 The process structure of a concurrent, connection-oriented server. A master server process accepts each incoming connection, and creates a slave process to handle it.

To avoid using CPU resources while it waits for connections, the master server uses a blocking call of *accept* to obtain the next connection from the well-known port. Thus, like the iterative server process in Chapter 10, the master server process in a con current server spends most of its time blocked in a call to *accept.* When a connection request arrives, the call to *accept* returns, allowing the master process to execute. The master creates a slave to handle the request, and reissues the call to *accept.* The call blocks the server again until another connection request arrives.

11.5 An Example Concurrent ECHO Server

File *TCPechod. c* contains the code for an ECHO server that uses concurrent processes to provide concurrent service to multiple clients.

```
/* TCPechod.c - main, TCPechod */


#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>


#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


#define QLEN           5    /* maximum connection queue length  */
#define BUFSIZE     4096
```

```
extern int  errno;

void    reaper(int);
int TCPechod(int fd);
int errexit(const char *format, ...);
int passiveTCP(const char *service, int qlen);

/*------------------------------------------------------------------------
 * main - Concurrent TCP server for ECHO service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *service = "echo";  /* service name or port number  */
    struct  sockaddr_in fsin;   /* the address of a client  */
    int alen;           /* length of client's address   */
    int msock;          /* master server socket     */
    int ssock;          /* slave server socket      */

    switch (argc) {
    case    1:
        break;
    case    2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPechod [port]\n");
    }

    msock = passiveTCP(service, QLEN);

    (void) signal(SIGCHLD, reaper);

    while (1) {
        alen = sizeof(fsin);
        ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
        if (ssock < 0) {
            if (errno == EINTR)
                continue;
            errexit("accept: %s\n", strerror(errno));
```

```
        }
        switch (fork()) {
        case 0:     /* child */
            (void) close(msock);
            exit(TCPechod(ssock));
        default:    /* parent */
            (void) close(ssock);
            break;
        case -1:
            errexit("fork: %s\n", strerror(errno));
        }
    }
}

/*------------------------------------------------------------------------
 * TCPechod - echo data until end of file
 *------------------------------------------------------------------------
 */
int
TCPechod(int fd)
{
    char    buf[BUFSIZ];
    int cc;

    while (cc = read(fd, buf, sizeof buf)) {
        if (cc < 0)
            errexit("echo read: %s\n", strerror(errno));
        if (write(fd, buf, cc) < 0)
            errexit("echo write: %s\n", strerror(errno));
    }
    return 0;
}

/*------------------------------------------------------------------------
 * reaper - clean up zombie children
 *------------------------------------------------------------------------
 */
/*ARGSUSED*/
void
reaper(int sig)
{
```

```
        int status;

        while (wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
            /* empty */;
}
```

As the example shows, the calls that control concurrency occupy only a small portion of the code. A master server process begins executing at *main.* After it checks its arguments, the master server calls *passiveTCP* to create a passive socket for the wellknown protocol port. It then enters an infinite loop.

During each iteration of the loop, the master server calls *accept* to wait for a connection request from a client. As in the iterative server, the call blocks until a request arrives. After the underlying TCP protocol software receives a connection request, the system creates a socket for the new connection, and the call to *accept* returns the socket descriptor.

After *accept* returns, the master server creates a slave process to handle the connection. To do so, the master process calls *fork* to divide itself into two processes[2]. The newly created child process first closes the master socket, and then calls procedure *TCPechod* to handle the connection. The parent process closes the socket that was created to handle the new connection, and continues executing the infinite loop. The next iteration of the loop will wait at the *accept* call for another new connection to arrive. Note that both the original and new processes have access to open sockets after the call to *fork(),* and that they both must close a socket before the system deallocates it. Thus, when the master process calls *close* for the new connection, the socket for that connection only disappears from the master process. Similarly, when the slave process calls *close* for the master socket, the socket only disappears from the slave process. The slave process continues to retain access to the socket for the new connection until the slave exits; the master server continues to retain access to the socket that corresponds to the well-known port.

After the slave closes the master socket, it calls procedure *TCPechod,* which provides the ECHO service for one connection. Procedure *TCPechod* consists of a loop that repeatedly calls *read* to obtain data from the connection and then calls *write* to send the same data back over the connection. Normally, *read* returns the (positive) count of bytes read. It returns a value less than zero if an error occurs (e.g., the network connection between the client and server breaks) or zero if it encounters an *end-of-file* condition (i.e., no more data can be extracted from the socket). Similarly, *write* normally returns the count of characters written, but returns a value less than zero if an error occurs. The slave checks the return codes, and uses *errexit* to print a message if an error occurs.

TCPechod returns zero if it can echo all data without error. When *TCPechod* returns, the main program uses the returned value as the argument in a call to *exit.* UNIX interprets the *exit* call as a request to terminate the process, and uses the argument as a process exit code. By convention, a process uses exit code zero to denote normal termination. Thus, the slave process exits normally after performing the ECHO service. When the slave exits, the system automatically closes all open descriptors, including the descriptor for the TCP connection.

11.6 Cleaning Up Errant Processes

Because concurrent servers generate processes dynamically, they introduce a potential problem of incompletely terminated processes. UNIX solves the problem by sending a *signal* to the parent whenever a child process exits. The exiting process remains in a *zombie* state until the parent executes a *wait3* system call. To terminate a child completely (i.e., to eliminate a zombie process), our example ECHO server catches the child termination signal and executes a signal handling function. The call

```
signal (SID, reaper) ;
```

informs the operating system that the master server process should execute function *reaper* whenever it receives a signal that a child process has exited (signal *SIGCHLD).* After the call to *signal,* the system automatically invokes *reaper* each time the server process receives a *SIGCHLD* signal.

[2] Recall from Chapter 3 that fork creates two processes, both executing the same code. The return value distinguishes between the original parent process and the newly created child.

Function *reaper* calls system function *wait3* to complete termination for a child that exits. *Wait3* blocks until one or more children exit (for any reason). It returns a value in the status structure that can be examined to find out about the process that exit ed. Because the program calls *wait3* when a *SIGCHLD* signal arrives, it will always be called after a child has exited. To ensure that an erroneous call does not deadlock the server, the program uses argument *WNOHANG* to specify that *wait3* should not block waiting for a process to exit, but should return immediately, even if no process has exited.

11.7 Summary

Connection-oriented servers achieve concurrency by allowing multiple clients to communicate with the server. The straightforward implementation in this chapter uses the fork function to create a new slave process each time a connection arrives. The master process never interacts with any clients; it merely accepts connections and creates a slave to handle each of them.

Each slave process begins execution in the main program immediately following the call to *fork.* The master process closes its copy of the descriptor for the new connection, and the slave closes its copy of the master descriptor. A connection to a client terminates after the slave exits because the operating system closes the slave's copy of the socket.

FOR FURTHER STUDY

Postel [RFC 862] defines the ECHO protocol used in the example TCP server.

EXERCISES

11.1 Instrument the server so it keeps a log of the time at which it creates each slave process and the time at which the slave terminates. How many clients must you start before you can find any overlap between the slave processes?

11.2 How many clients can access the example concurrent server simultaneously before any client must be denied service? How many can access the iterative server in Chapter 10 before any is denied service?

11.3 Build an iterative implementation of an ECHO server. Conduct an experiment to determine if a human can sense the difference in response time between the concurrent and iterative versions.

11.4 Modify the example server so procedure *TCPechod* explicitly closes the connection before it returns. Explain why an explicit call to *close* might make the code easier to maintain.

12
Single-Process, Concurrent Servers (TCP)

12.1 Introduction

The previous chapter illustrates how most concurrent, connection-oriented servers operate. They use operating system facilities to create a separate process for each connection, and allow the operating system to timeslice the processor among the processes. This chapter illustrates a design idea that is interesting, but not obvious: it shows how a server can offer apparent concurrency to clients while using only a single process. First, it examines the general idea. It discusses why such an approach is feasible and when it may be superior to an implementation using multiple processes. Second, it considers how a single-process server uses the UNIX system calls to handle multiple connections concurrently. The example server follows Algorithm 8.5[1].

12.2 Data-driven Processing In A Server

For applications where I/O dominates the cost of preparing a response to a request, a server can use asynchronous I/O to provide apparent concurrency among clients. The idea is simple: arrange for a single server process to keep TCP connections open to multiple clients, and have the server handle a given connection when data arrives. Thus, the server uses the arrival of data to trigger processing.

To understand why the approach works, consider the concurrent ECHO server described in the previous chapter. To achieve concurrent execution, the server creates a separate slave process to handle each new connection. In theory, the server depends on the operating system's timeslicing mechanism to share the CPU among the processes, and hence, among the connections.

In practice, however, an ECHO server seldom depends on timeslicing. If one were able to watch the execution of a concurrent ECHO server closely, one would find thal the arrival of data often controls processing. The reason relates to data flow across an internet. Data arrives at the server in bursts, not in a steady stream, because the underlying internet delivers data in discrete packets. Clients add to the bursty behavior ii they choose to send blocks of data so that the resulting TCP segments each fit into a single IP datagram. At the server, each slave process spends most of its time blocked in a call to *read* waiting for the next burst to arrive. Once the data arrives, the *read* call returns and the slave process executes. The slave calls *write* to send the data back tc the client, and then calls *read* again to wait for more data. A CPU that can handle the load of many clients without slowing down must execute sufficiently fast to complete the cycle of reading and writing before data arrives for another slave.

Of course, if the load becomes so great that the CPU cannot process one requesl before another arrives, timesharing takes over. The operating system switches the processor among all slaves that have data to process. For simple services that require little processing for each request, chances are high that execution will be driven by the arrival of data. To summarize:

> *Concurrent servers that require little processing time per request often behave in a sequential manner where the arrival of data triggers execution. Timesharing only takes over if the load becomes so high that the CPU cannot handle it sequentially.*

12.3 Data-Driven Processing With A Single Process

Understanding the sequential nature of a concurrent server's behavior allows us to understand how a single process can perform the same task. Imagine a single server process that has TCP connections open to many clients. The process blocks waiting for data to arrive. As soon as data arrives on any connection, the process awakens, handles the request, and sends a reply. It then blocks again, waiting for more data to arrive from another connection. As long as the CPU is fast enough to satisfy the load presented to the server, the single process version handles requests as well as a version with multiple processes. In fact,

---

[1] See Section 8.23 for a description of Algorithm 8.5.

because a single-process implementation requires less switching between process contexts, it may be able to handle a slightly higher load than an implementation that uses multiple processes.

The key to programming a single-process concurrent server lies in the use of asynchronous I/O through the operating system primitive *select.* A server creates a socket for each of the connections it must manage, and then calls *select* to wait for data to ar rive on any of them. In fact, because *select* can wait for I/O on all possible sockets, it can also wait for new connections at the same time. Algorithm 8.5 lists the detailed steps a single-process server uses.

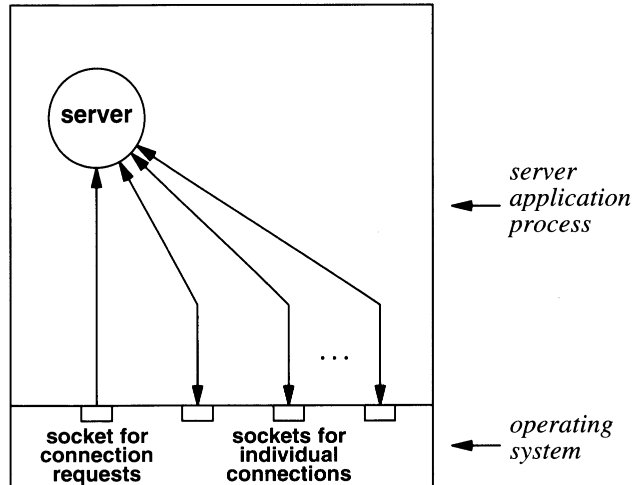

Figure 12.1 The process structure of a connection-oriented server that achieves concurrency with a single process. The process manages multiple sockets.

12.4 Process Structure Of A Single-Process Server

Figure 12.1 illustrates the process and socket structure of a single-process, concurrent server. One process manages all sockets.

In essence, a single process server must perform the duties of both the master and slave processes. It maintains a set of sockets, with one socket in the set bound to the well-known port at which the master would accept connections. The other sockets in the set each correspond to a connection over which a slave would handle requests. The server passes the set of socket descriptors as an argument to *select,* and waits for activity on any of them. When *select* returns, it passes back a bit mask that specifies which of the descriptors in the set is ready. The server uses the order in which descriptors become ready to decide how to proceed.

To distinguish between master and slave operations, a single-process server uses the descriptor. If the descriptor that corresponds to the master socket becomes ready, the server performs the same operation the master would perform: it calls *accept* on the socket to obtain a new connection. If a descriptor that corresponds to a slave socket becomes ready, the server performs the operation a slave would perform: it calls *read* to obtain a request, and then answers it.

12.5 An Example Single-Process ECHO Server

An example will help clarify the ideas and explain how a single-process, concurrent server works. Consider file *TCPmechod.c.* which contains the code for a single-process server that implements the ECHO service.

```
/* TCPmechod.c - main, echo */


#include <sys/types.h>

#include <sys/socket.h>

#include <sys/time.h>
```

```
#include <netinet/in.h>

#include <unistd.h>
#include <string.h>
#include <stdio.h>

#define QLEN           5    /* maximum connection queue length  */
#define BUFSIZE     4096

extern int  errno;
int     errexit(const char *format, ...);
int     passiveTCP(const char *service, int qlen);
int     echo(int fd);

/*------------------------------------------------------------------------
 * main - Concurrent TCP server for ECHO service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *service = "echo";  /* service name or port number  */
    struct sockaddr_in fsin;    /* the from address of a client */
    int msock;          /* master server socket     */
    fd_set  rfds;           /* read file descriptor set */
    fd_set  afds;           /* active file descriptor set   */
    int alen;           /* from-address length      */
    int fd, nfds;

    switch (argc) {
    case    1:
        break;
    case    2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPmechod [port]\n");
    }

    msock = passiveTCP(service, QLEN);
```

```
    nfds = getdtablesize();
    FD_ZERO(&afds);
    FD_SET(msock, &afds);

    while (1) {
        memcpy(&rfds, &afds, sizeof(rfds));

        if (select(nfds, &rfds, (fd_set *)0, (fd_set *)0,
                (struct timeval *)0) < 0)
            errexit("select: %s\n", strerror(errno));
        if (FD_ISSET(msock, &rfds)) {
            int ssock;

            alen = sizeof(fsin);
            ssock = accept(msock, (struct sockaddr *)&fsin,
                &alen);
            if (ssock < 0)
                errexit("accept: %s\n",
                    strerror(errno));
            FD_SET(ssock, &afds);
        }
        for (fd=0; fd<nfds; ++fd)
            if (fd != msock && FD_ISSET(fd, &rfds))
                if (echo(fd) == 0) {
                    (void) close(fd);
                    FD_CLR(fd, &afds);
                }
    }
}

/*------------------------------------------------------------------------
 * echo - echo one buffer of data, returning byte count
 *------------------------------------------------------------------------
 */
int
echo(int fd)
{
    char    buf[BUFSIZ];
    int cc;

    cc = read(fd, buf, sizeof buf);
```

```
    if (cc < 0)
        errexit("echo read: %s\n", strerror(errno));
    if (cc && write(fd, buf, cc) < 0)
        errexit("echo write: %s\n", strerror(errno));
    return cc;
}
```

The single-process server begins, like the master server in a multiple-process implementation, by opening a passive socket at the well-known port. It uses *FD_ZERO* and *FD_SET* to create a bit vector that corresponds to the socket descriptors that it wishes to test. The server then enters an infinite loop in which it calls *select* to wait for one or more of the descriptors to become ready.

If the master descriptor becomes ready, the server calls *accept* to obtain a new connection. It adds the descriptor for the new connection to the set it manages, and continues to wait for more activity. If a slave descriptor becomes ready, the server calls pro cedure *echo* which calls *read* to obtain data from the connection and *write* to send it back to the client. If one of the slave descriptors reports an end-of-file condition, the server closes the descriptor and uses macro *FD_CLR* to remove it from the set of descriptors *select* uses.

12.6 Summary

Execution in concurrent servers is often driven by the arrival of data and not by the timeslicing mechanism in the underlying operating system. In cases where the service requires little processing, a single-process implementation can use asynchronous I/O to manage connections to multiple clients as effectively as an implementation that uses multiple processes.

The single-process implementation performs the duties of the master and slave processes. When the master socket becomes ready, the server accepts a new connection. When any other socket becomes ready, the server reads a request and sends a re ply. An example single-process server for the ECHO service illustrates the ideas and shows the programming details.

FOR FURTHER STUDY

A good protocol specification does not constrain the implementation. For example, the single-process server described in this chapter implements the ECHO protocol defined by Postel [RFC 862]. Chapter 11 shows an example of a multiple-process, concurrent server built from the same protocol specification.

EXERCISES

12.1 Conduct an experiment that proves the example ECHO server can handle connections concurrently.

12.2 Does it make sense to use the implementation discussed in this chapter for the DAYTIME service? Why or why not?

12.3 Read the *UNIX Programmer's Manual* to find out the exact representation of descriptors in the list passed to *select*. Write the *FD_SET* and *FD_CLR* macros.

12.4 Compare the performance of single-process and multiple-process server implementations on a computer with multiple processors. Under what circumstances will a single-process version perform better than (or equal to) a multiple-process version?

12.5 Suppose a large number of clients (e.g., 100) access the example server in this chapter at the same time. Explain what each client might observe.

12.6 Can a single-process server ever deprive one client of service while it repeatedly honors requests from another? Can a multiple-process implementation ever exhibit the same behavior? Explain.

12
Single-Process, Concurrent Servers (TCP)

12.1 Introduction

The previous chapter illustrates how most concurrent, connection-oriented servers operate. They use operating system facilities to create a separate process for each connection, and allow the operating system to timeslice the processor among the processes. This chapter illustrates a design idea that is interesting, but not obvious: it shows how a server can offer apparent concurrency to clients while using only a single process. First, it examines the general idea. It discusses why such an approach is feasible and when it may be superior to an implementation using multiple processes. Second, it considers how a single-process server uses the UNIX system calls to handle multiple connections concurrently. The example server follows Algorithm 8.5[1].

12.2 Data-driven Processing In A Server

For applications where 1/O dominates the cost of preparing a response to a request, a server can use asynchronous I/O to provide apparent concurrency among clients. The idea is simple: arrange for a single server process to keep TCP connections open to multiple clients, and have the server handle a given connection when data arrives. Thus, the server uses the arrival of data to trigger processing.

To understand why the approach works, consider the concurrent ECHO server described in the previous chapter. To achieve concurrent execution, the server creates a separate slave process to handle each new connection. In theory, the server depends on the operating system's timeslicing mechanism to share the CPU among the processes, and hence, among the connections.

In practice, however, an ECHO server seldom depends on timeslicing. If one were able to watch the execution of a concurrent ECHO server closely, one would find that the arrival of data often controls processing. The reason relates to data flow across an internet. Data arrives at the server in bursts, not in a steady stream, because the underlying internet delivers data in discrete packets. Clients add to the bursty behavior if they choose to send blocks of data so that the resulting TCP segments each fit into a single IP datagram. At the server, each slave process spends most of its time blocked in a call to *read* waiting for the next burst to arrive. Once the data arrives, the *read* call returns and the slave process executes. The slave calls *write* to send the data back to the client, and then calls *read* again to wait for more data. A CPU that can handle the load of many clients without slowing down must execute sufficiently fast to complete the cycle of reading and writing before data arrives for another slave.

Of course, if the load becomes so great that the CPU cannot process one request before another arrives, timesharing takes over. The operating system switches the processor among all slaves that have data to process. For simple services that require little processing for each request, chances are high that execution will be driven by the arrival of data. To summarize:

> *Concurrent servers that require little processing time per request often behave in a sequential manner where the arrival of data triggers execution. Timesharing only takes over if the load becomes so high that the CPU cannot handle it sequentially.*

12.3 Data-Driven Processing With A Single Process

Understanding the sequential nature of a concurrent server's behavior allows us to understand how a single process can perform the same task. Imagine a single server process that has TCP connections open to many clients. The process blocks waiting for data to arrive. As soon as data arrives on any connection, the process awakens, handles the request, and sends a reply. It then blocks again, waiting for more data to arrive from another connection. As long as the CPU is fast enough to satisfy the load presented to the server, the single process version handles requests as well as a version with multiple processes. In fact,

---

[1] See page Section 8.23 for a description of Algorithm 8.5.

because a single-process implementation requires less switching between process contexts, it may be able to handle a slightly higher load than an implementation that uses multiple processes.

The key to programming a single-process concurrent server lies in the use of asynchronous 1/O through the operating system primitive *select.* A server creates a socket for each of the connections it must manage, and then calls *select* to wait for data to ar rive on any of them. In fact, because *select* can wait for 1/O on all possible sockets, it can also wait for new connections at the same time. Algorithm 8.5 lists the detailed steps a single-process server uses.

12.4 Process Structure Of A Single-Process Server

Figure 12.1 illustrates the process and socket structure of a single-process, concurrent server. One process manages all sockets.

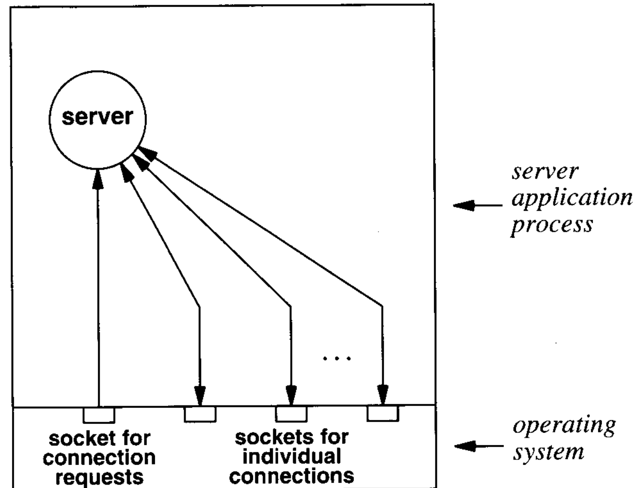

Figure 12.1 The process structure of a connection-oriented server that achieves concurrency with a single process. The process manages multiple sockets.

In essence, a single process server must perform the duties of both the master and :lave processes. It maintains a set of sockets, with one socket in the set bound to the well-known port at which the master would accept connections. The other sockets in the set each correspond to a connection over which a slave would handle requests. The server passes the set of socket descriptors as an argument to *select,* and waits for activity on any of them. When *select* returns, it passes back a bit mask that specifies which of the descriptors in the set is ready. The server uses the order in which descriptors become ready to decide how to proceed.

To distinguish between master and slave operations, a single-process server uses the descriptor. If the descriptor that corresponds to the master socket becomes ready, the server performs the same operation the master would perform: it calls *accept* on the socket to obtain a new connection. If a descriptor that corresponds to a slave socket becomes ready, the server performs the operation a slave would perform: it calls *read* to obtain a request, and then answers it.

12.5 An Example Single-Process ECHO Server

An example will help clarify the ideas and explain how a single-process, concurrent server works. Consider file *TCPmechod.c,* which contains the code for a single-process server that implements the ECHO service.

```
/* TCPmechod.c - main, echo */


#include <sys/types.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <netinet/in.h>
```

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>

#define QLEN           5    /* maximum connection queue length  */
#define BUFSIZE     4096

extern int  errno;
int     errexit(const char *format, ...);
int     passiveTCP(const char *service, int qlen);
int     echo(int fd);

/*------------------------------------------------------------------------
 * main - Concurrent TCP server for ECHO service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
    char    *service = "echo";  /* service name or port number  */
    struct sockaddr_in fsin;    /* the from address of a client */
    int msock;          /* master server socket     */
    fd_set  rfds;           /* read file descriptor set */
    fd_set  afds;           /* active file descriptor set   */
    int alen;           /* from-address length      */
    int fd, nfds;

    switch (argc) {
    case    1:
        break;
    case    2:
        service = argv[1];
        break;
    default:
        errexit("usage: TCPmechod [port]\n");
    }

    msock = passiveTCP(service, QLEN);

    nfds = getdtablesize();
```

```
    FD_ZERO(&afds);
    FD_SET(msock, &afds);

    while (1) {
        memcpy(&rfds, &afds, sizeof(rfds));

        if (select(nfds, &rfds, (fd_set *)0, (fd_set *)0,
                (struct timeval *)0) < 0)
            errexit("select: %s\n", strerror(errno));
        if (FD_ISSET(msock, &rfds)) {
            int ssock;

            alen = sizeof(fsin);
            ssock = accept(msock, (struct sockaddr *)&fsin,
                &alen);
            if (ssock < 0)
                errexit("accept: %s\n",
                    strerror(errno));
            FD_SET(ssock, &afds);
        }
        for (fd=0; fd<nfds; ++fd)
            if (fd != msock && FD_ISSET(fd, &rfds))
                if (echo(fd) == 0) {
                    (void) close(fd);
                    FD_CLR(fd, &afds);
                }
    }
}

/*------------------------------------------------------------------------
 * echo - echo one buffer of data, returning byte count
 *------------------------------------------------------------------------
 */
int
echo(int fd)
{
    char    buf[BUFSIZ];
    int cc;

    cc = read(fd, buf, sizeof buf);
    if (cc < 0)
```

```
        errexit("echo read: %s\n", strerror(errno));
    if (cc && write(fd, buf, cc) < 0)
        errexit("echo write: %s\n", strerror(errno));
    return cc;
}
```

The single-process server begins, like the master server in a multiple-process implementation, by opening a passive socket at the well-known port. It uses *FD_ZERO* and *FD_SET* to create a bit vector that corresponds to the socket descriptors that it wishes to test. The server then enters an infinite loop in which it calls *select* to wait for one or more of the descriptors to become ready.

If the master descriptor becomes ready, the server calls *accept* to obtain a new connection. It adds the descriptor for the new connection to the set it manages, and continues to wait for more activity. If a slave descriptor becomes ready, the server calls pro cedure *echo* which calls *read* to obtain data from the connection and *write* to send it back to the client. If one of the slave descriptors reports an end-of-file condition, the server closes the descriptor and uses macro *FD_CLR* to remove it from the set of descriptors *select* uses.

12.6 Summary

Execution in concurrent servers is often driven by the arrival of data and not by the timeslicing mechanism in the underlying operating system. In cases where the service requires little processing, a single-process implementation can use asynchronous I/O to manage connections to multiple clients as effectively as an implementation that uses multiple processes.

The single-process implementation performs the duties of the master and slave processes. When the master socket becomes ready, the server accepts a new connection. When any other socket becomes ready, the server reads a request and sends a reply. An example single-process server for the ECHO service illustrates the ideas and shows the programming details.

FOR FURTHER STUDY

A good protocol specification does not constrain the implementation. For example, the single-process server described in this chapter implements the ECHO protocol defined by Postel [RFC 862]. Chapter 11 shows an example of a multiple-process, con- ;urrent server built from the same protocol specification.

EXERCISES

12.1 Conduct an experiment that proves the example ECHO server can handle connections concurrently.

12.2 Does it make sense to use the implementation discussed in this chapter for the DAYTIME service? Why or why not?

12.3 Read the *UNIX Programmer's Manual* to find out the exact representation of descriptors in the list passed to *select*. Write the *FD SET* and *FD _CLR* macros.

12.4 Compare the performance of single-process and multiple-process server implementations on a computer with multiple processors. Under what circumstances will a single-process version perform better than (or equal to) a multiple-process version?

12.5 Suppose a large number of clients (e.g., 100) access the example server in this chapter at the same time. Explain what each client might observe.

11.6 Can a single-process server ever deprive one client of service while it repeatedly honors requests from another? Can a multiple-process implementation ever exhibit the same behavior? Explain.