UNIX® Network Programming

W. Richard Stevens

Health Systems International

PTR Prentice Hall
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Stevens, W. Richard.
UNIX network programming / W. Richard Stevens.
p. cm.
Includes bibliographical references.
ISBN 0-13-949876-1
1. UNIX (Computer operating system) 2. Computer networks.
I. Title.
QA76.76.063S755 1990
005.7'1--dc20 89-25576
CIP

To Sally, Bill, and Ellen.

Editorial/production supervision: **Brendan M. Stewart**
Cover design: **Lundgren Graphics Ltd.**
Manufacturing buyer: **Ray Sintel**

Prentice Hall Software Series
Brian W. Kernighan, Advisor

Printed in the United States of America
20 19 18 17 16 15 14 13 12 11

ISBN 0-13-949876-1

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

1

Introduction

1.1 History

Computer networks have revolutionized our use of computers. They pervade our every-day life, from automated teller machines, to airline reservation systems, to electronic mail services, to electronic bulletin boards. There are many reasons for the explosive growth in computer networks.

- The proliferation of personal computers and workstations during the 1980s helped fuel the interest and need for networks.
- Computer networks used to be expensive and were restricted to large universities, government research sites, and large corporations. Technology has greatly reduced the cost of establishing computer networks, and networks are now found in organizations of every size.
- Many computer manufacturers now package networking software as part of the basic operating system. Networking software is no longer regarded as an add-on that only a few customers will want. It is now considered as essential as a text editor.
- We are in an information age and computer networks are becoming an integral part in the dissemination of information.

Computer systems used to be stand-alone entities. Each computer was self-contained and had all the peripherals and software required to do a particular job. If a particular feature was needed, such as line printer output, a line printer was attached to

the system. If large amounts of disk storage were needed, disks were added to the system. What helped change this is the realization that computers and their users need to *share* information and resources.

Information sharing can be electronic mail or file transfer. Resource sharing can involve accessing a peripheral on another system. Twenty years ago this type of sharing took place by exchanging magnetic tapes, decks of punched cards, and line printer listings. Today computers can be connected together using various electronic techniques called *networks*. A network can be as simple as two personal computers connected together using a 1200 baud modem, or as complex as the TCP/IP Internet, which connects over 150,000 systems together. The number of ways to connect a computer to a network are many, as are the various things we can do once connected to a network. Some typical network applications are

- Exchange electronic mail with users on other computers. It is commonplace these days for people to communicate regularly using electronic mail.
- Exchange files between systems. For many applications it is just as easy to distribute the application electronically, instead of mailing diskettes or magnetic tapes. File transfer across a network also provides faster delivery.
- Share peripheral devices. Examples range from the sharing of line printers to the sharing of magnetic tape drives. A large push towards the sharing of peripheral devices has come from the personal computer and workstation market, since often the cost of a peripheral can exceed the cost of the computer. In an organization with many personal computers or workstations, sharing peripherals makes sense.
- Execute a program on another computer. There are cases where some other computer is better suited to run a particular program. For example, a time-sharing system or a workstation with good program development tools might be the best system on which to edit and debug a program. Another system, however, might be better equipped to run the program. This is often the case with programs that require special features, such as parallel processing or vast amounts of storage. The National Science Foundation (NSF) has connected the six NSF supercomputer centers using a network, allowing scientists to access these computers electronically. Years ago, access to facilities such as this would have required mailing decks of punched cards or tapes.
- Remote login. If two computers are connected using a network, we should be able to login to one from the other (assuming we have an account on both systems). It is usually easier to connect computers together using a network, and provide a remote login application, than to connect every terminal in an organization to every computer.

We'll consider each of these applications in more detail in the later chapters of this text.

1.2 Layering

Given a particular task that we want to solve, such as providing a way to exchange files between two computers that are connected with a network, we divide the task into pieces and solve each piece. In the end we connect the pieces back together to form the final solution. We could write a single monolithic system to solve the problem, but experience has shown that solving the problem in pieces leads to a better, and more extensible, solution.

It is possible that part of the solution developed for a file transfer program can also be used for a remote printing program. Also, if we're writing the file transfer program assuming the computers are connected with an Ethernet, it could turn out that part of this program is usable for computers connected with a leased telephone line.

In the context of networking, this is called *layering*. We divide the communication problem into pieces (layers) and let each layer concentrate on providing a particular function. Well-defined interfaces are provided between layers.

1.3 OSI Model

The starting point for describing the layers in a network is the International Standards Organization (ISO) *open systems interconnection* model (OSI) for computer communications. This is a 7-layer model shown in Figure 1.1.

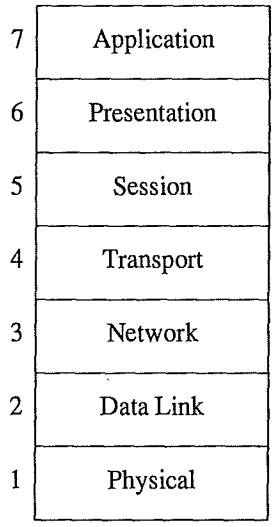


Figure 1.1 OSI 7-layer model.

The OSI model provides a detailed standard for describing a network. Most computer networks today are described using the OSI model. In Chapter 5, each of the networks we describe is shown against the OSI model.

The transport layer is important because it is the lowest of the seven layers that provides reliable data transfer between two systems. The layers above the transport layer can assume they have an error-free communications path. Details such as sequence control, error detection, retransmission and flow control are handled by the transport layer and the layers below it.

1.4 Processes

A fundamental entity in a computer network is a *process*. A process is a program that is being executed by the computer's operating system. When we say that two computers are communicating with each other, we mean that two processes, one running on each computer, are in communication with each other.

We describe processes in the Unix operating system in detail in Chapter 2. In Chapter 3 we describe the various forms of interprocess communication (IPC) provided by Unix to allow processes on a single computer to exchange information. In Chapters 6 and 7 we expand the interprocess communication to include processes on different systems that are connected by a network.

1.5 A Simplified Model

This book is mainly concerned with the top three layers of the OSI model—the session, presentation, and application layers—along with the interface between these three layers and the transport layer beneath them. We'll often combine these three layers into one and call it the *process* layer. Additionally, we'll usually combine the bottom two layers into a single layer and call it the *data-link* layer. We'll use a simplified, 4-layer version of the OSI model, illustrated in Figure 1.2.

In Figure 1.2 we show two systems that are connected with a network. The actual connection between the two systems is between the two data-link layers, the solid horizontal line. Although it appears to the two processes that they are communicating with each other (the dashed horizontal line between the two process layers), the actual data flows from the process layer, down to the transport layer, down to the network layer, down to the data-link layer, across the physical network to the other data-link layer, up through the network layer, then the transport layer to the other process.

Layering leads to the definitions of *protocols* at each layer. Even though physical communication takes place at the lowest layer (the physical layer in the OSI model), protocols exist at every layer. The protocols that are used at a given layer, the four horizontal lines in Figure 1.2, are also called *peer-to-peer protocols* to reiterate that they are used between two entities at the same layer. We'll discuss various protocols at the data-link, network, and transport layers, in Chapter 5. The later chapters, which describe specific applications, present various application-specific protocols at the process layer.

Since the data always flows between a given layer and the layer immediately above it or immediately below it, the *interfaces* between the layers are also important for

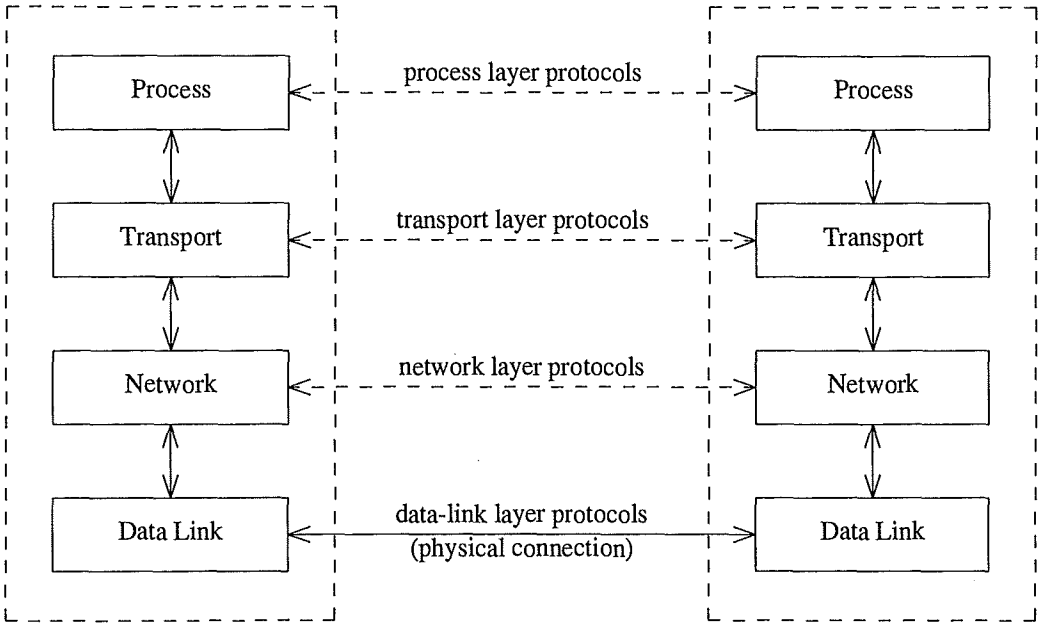


Figure 1.2 Simplified 4-layer model.

network programming. Chapters 6 and 7 provide detailed descriptions of two interfaces between the transport layer and the layer above it: Berkeley sockets and the AT&T Transport Layer Interface (TLI). The Berkeley socket interface is then used in the remaining chapters for most of the examples.

Another feature of the model shown in Figure 1.2 is that often the layers up through and including the transport layer (layers 1 through 4 of the OSI model) are included in the host operating system. This is the case for 4.3BSD and System V, the two examples used in this text.

1.6 Client–Server Model

The standard model for network applications is the *client–server* model. A *server* is a process that is waiting to be contacted by a *client* process so that the server can do something for the client. A typical (but not mandatory) scenario is as follows:

- The server process is started on some computer system. It initializes itself, then goes to sleep waiting for a client process to contact it requesting some service.
- A client process is started, either on the same system or on another system that is connected to the server's system with a network. Client processes are often initiated by an interactive user entering a command to a time-sharing system. The client process sends a request across the network to the server requesting service of some form. Some examples of the type of service that a server can provide are

- return the time-of-day to the client,
- print a file on a printer for the client,
- read or write a file on the server's system for the client,
- allow the client to login to the server's system,
- execute a command for the client on the server's system.

- When the server process has finished providing its service to the client, the server goes back to sleep, waiting for the next client request to arrive.

We can further divide the server processes into two types.

1. When a client's request can be handled by the server in a known, short amount of time, the server process handles the request itself. We call these *iterative servers*. A time-of-day service is typically handled in an iterative fashion by the server.
2. When the amount of time to service a request depends on the request itself (so that the server doesn't know ahead of time how much effort it takes to handle each request), the server typically handles it in a concurrent fashion. These are called *concurrent servers*. A concurrent server invokes another process to handle each client request, so that the original server process can go back to sleep, waiting for the next client request. Naturally, this type of server requires an operating system that allows multiple processes to run at the same time. (More on this in the next chapter.) Most client requests that deal with a file of information (print a file, read or write a file, for example) are handled in a concurrent fashion by the server, as the amount of processing required to handle each request depends on the size of the file.

We'll see examples of both types of servers in later chapters.

1.7 Plan of the Book

The goal of the book is to develop and present examples of processes that are used for communication between different computer systems. In Figure 1.3, the chapters in this book are compared to the OSI model.

To develop applications at the process layer, a thorough understanding of a process is required. Chapter 2 develops the process terminology for the Unix system. The emphasis is on process control and signals. Chapter 3 describes methods by which multiple processes on a single computer can communicate with each other—interprocess communication, or IPC. Understanding how independent processes communicate on a single computer system is a requisite to understanding IPC between processes on different systems. Chapter 3 also describes file locking and record locking under Unix, as these are often required when multiple processes are used for a task.

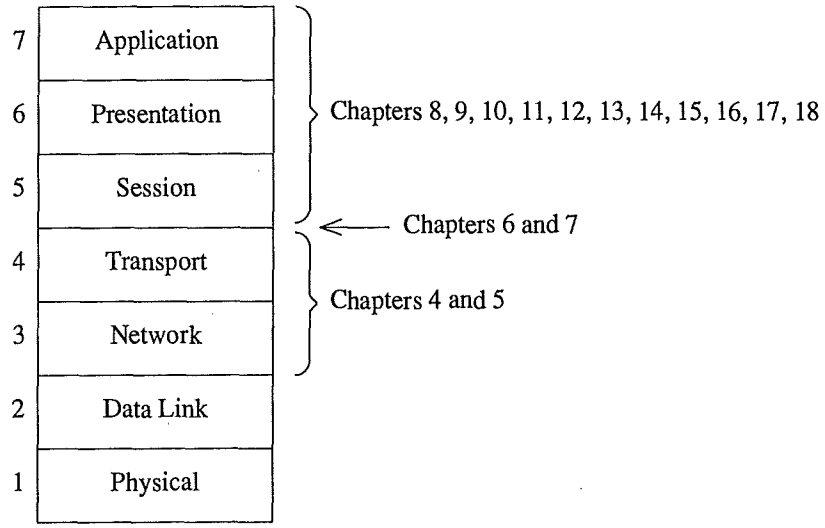


Figure 1.3 Outline of book compared to 7-layer OSI model.

Chapter 4 provides an overview of networking and the terminology used to describe the services provided by a network. Various communication protocol suites are described in Chapter 5. The emphasis is on the services provided to an application process by the protocol suite. We concentrate on the TCP/IP and Xerox NS (XNS) protocol suites, since they are used in the examples in the later chapters. We also cover the following protocol suites: IBM's SNA, NetBIOS, the OSI protocols, and UUCP. SNA is important since most Unix vendors today provide support for SNA. NetBIOS is used in many personal computer networks, which Unix systems need to communicate with. The OSI protocols, while still in their infancy, will be more important in the coming decade. An overview of UUCP is provided since it is the oldest of the Unix networking solutions, and is still widely used. An understanding of where these different protocol suites fit with regard to each other is important.

Chapters 6 and 7 describe the two predominant interfaces provided by Unix today, between the transport layer and an application process: Berkeley sockets and System V TLI. The Berkeley socket interface is then used for most of the examples in the later chapters.

The final eleven chapters contain detailed examples of actual process-level application programs. Chapter 8 describes various utility functions that are used in the remaining chapters. This includes converting network names into addresses, and timeout and retransmission algorithms for unreliable protocols.

Security is an issue we'll encounter in almost every application, and we cover it in Chapter 9. We describe the security techniques currently used by the 4.3BSD system along with the newer Kerberos system.

Chapter 10 is the first chapter describing a specific network application: obtaining the time and date from another computer on the network. Chapter 11 describes the ubiquitous Internet "ping" program, which has become an everyday tool on most TCP/IP networks. We also present a similar application using the Xerox NS protocols.

Chapter 12 is a complete presentation of a file transfer program—the TCP/IP Trivial File Transfer Protocol (TFTP). File transfer is an important network application and TFTP is one of the simpler protocols.

Line printer spoolers are covered in Chapter 13, both the Berkeley and System V print spoolers. Print spoolers are interesting from a process control viewpoint, in addition to the network printing option provided by 4.3BSD.

Chapter 14 covers the execution of commands on another system. We concentrate on the `rcmd` function provided by 4.3BSD. This function is central to what are called the Berkeley "r" commands: `rcp`, `rdump`, `rlogin`, and the like. These programs are often provided by third-party TCP/IP vendors for systems other than 4.3BSD, and have become de facto standards for Unix networking applications.

Remote login across a network is the topic of Chapter 15. This chapter also concerns itself with details of terminal handling and pseudo-terminals, both important features in remote login programs. This chapter concludes with a complete presentation of the 4.3BSD `rlogin` client and server.

In Chapter 16 we describe the use of a tape drive across a network, specifically the 4.3BSD `rmt` protocol. This has become an important topic with workstations and personal computers that have disks but not tape drives.

Chapter 17 is concerned with performance. This includes the performance of the various IPC techniques from Chapter 3, along with the performance of a network. Since a network is often used for file transfer and tape drive access, typical performance figures for disk drives and tape drives are also covered.

Chapter 18 covers remote procedure calls (RPC), a technique used to simplify the programming of network applications. We cover three RPC implementations: Sun RPC, Xerox Courier, and Apollo NCA.

1.8 A History of Unix Networking

The first Unix network application was UUCP, the Unix-to-Unix copy program, and all its associated commands. UUCP was developed around 1976 [Redman 1989] and its first major release outside AT&T was with Version 7 Unix in 1978. UUCP is a batch-oriented system that is typically used between systems with dial-up telephone lines or systems that are directly connected. Its major uses are for distributing software (file transfer) and electronic mail. We'll say more about the programs that comprise UUCP in Section 5.7. It is interesting that the first Unix networking application is still in wide use today.

The program `cu` was also distributed with Version 7 Unix in 1978, and has been available with almost every Unix system since then. Although not strictly part of a specific networking package, it provides a remote login capability to another computer system. The `cu` program was typically used to connect to another system using a dial-up telephone line. A rewritten version of `cu` was distributed with 4.2BSD where it was called `tip`.

In 1978 Eric Schmidt developed a network application for Unix as part of his Masters degree at Berkeley. This network was called "Berknet" and was first distributed with the Berkeley release of Version 7 Unix for the PDP-11—the 2.xBSD systems. It provided file transfer, electronic mail, and remote printing. It was widely used on the Berkeley campus. Typically the systems were directly connected with 9600 baud RS-232 lines. This software moved to the VAX with the 4.xBSD releases, then it was slowly replaced with the faster Ethernet-based local area networks that started with the 4.2BSD system.

In September 1980 Bolt, Beranek, and Newman (BBN) was contracted by the Defense Advanced Research Projects Agency (DARPA) of the Department of Defense (DoD) to develop an implementation of the TCP/IP protocols for Berkeley Unix on the VAX [Walsh and Gurwitz 1984]. A version for the 4.1BSD system was turned over to Berkeley in the Fall of 1981. This was then integrated by Berkeley into the 4.2BSD system that was being developed. The 4.2BSD system was released in August 1983 and with it the growth of local area networks, based mainly on Ethernet technology, grew rapidly. These 4.2BSD systems also allowed the computers to connect to the ARPANET, which also saw enormous growth in the early 1980s. With the 4.2BSD system the socket interface was provided, which we describe in Chapter 6 and use for most of the examples in the remainder of the text.

Networking development during this period was also underway at AT&T, however, other than UUCP, the results usually did not propagate outside AT&T. This is in contrast to the networking developments at Berkeley, which were widely disseminated. Fritz, Hefner, and Raleigh [1984] describe a network based on a HYPERchannel high-speed local network. Its development started around 1980 and it connected various types of systems running different versions of Unix: VAX 780, PDP-11/70, IBM 370, and AT&T 3B20S. The versions of Unix were the AT&T versions of System III and System V. The network was batch-oriented and supported file transfer, remote command execution, remote printing, and electronic mail.

Various third-party networking packages appeared for System V in the mid-1980s. These typically supported the TCP/IP protocol suite and were often developed by the vendor who developed the interface hardware.

The streams I/O facility was first described in Ritchie [1984b]. It did not get widely distributed outside the Bell Labs research environment, however, until System V Release 3.0 in 1986. The Transport Layer Interface (TLI), which we describe in Chapter 7, was provided with this release of System V [Olander, McGrath, and Israel 1986]. Various third-party networking packages (mainly for the TCP/IP protocol suite) based on TLI started to appear around 1988.

4

A Network Primer

We have to define some basic networking terms and concepts before they are used in later chapters. When necessary, we jump ahead and use specific examples from the next chapter, to provide actual network examples, instead of trying to describe everything in a generic, abstract sense. Tanenbaum [1989] provides additional details on many of the networking topics discussed in this chapter.

Internetworking

A *computer network* is a communication system for connecting end-systems. We often refer to the end-systems as *hosts*. The hosts can range in size from small microcomputers to the largest supercomputers. Some hosts on a computer network are dedicated systems, such as print servers or file servers, without any capabilities for interactive users. Other hosts might be single-user personal computers, while others might be general purpose time-sharing systems.

A *local area network*, or *LAN*, connects computer systems that are close together—typically within a single building, but possibly up to a few kilometers apart. Popular technologies today for LANs are Ethernet and token ring. LANs typically operate at high speeds—an Ethernet operates at 10 Mbps (million bits per second) while IBM's token ring operates at both 4 and 16 Mbps. Newer LAN technologies, such as FDDI (Fiber Distributed Data Interface), use fiber optics and have a data rate of 100 Mbps. Each computer on a LAN has an interface card of some form that connects it to the actual network hardware. (Be aware that these raw network speeds are usually not realized in actual data transfers. In Chapter 17 we'll discuss some actual performance measurements.)

A *wide area network* or *WAN* connects computers in different cities or countries. These networks are sometimes referred to as *long haul networks*. A common technology for WANs is leased telephone lines operating between 9600 bps (bits per second) and 1.544 Mbps (million bits per second).

Between the LAN and WAN is the *metropolitan area network* or *MAN*. These cover an entire city or metropolitan area and frequently operate at LAN speeds. Common technologies are coaxial cable (similar to cable TV) and microwave.

An *internet* or *internetwork* is the connection of two or more distinct networks so that computers on one network are able to communicate with computers on another network. The goal of internetworking is to hide the details of what might be different physical networks, so that the internet functions as a coordinated unit. One way to connect two distinct physical networks is to have a *gateway* that is attached to both networks. This gateway must pass information from one network to the other. It is sometimes called a *router*. As an example, Figure 4.1 shows an internet formed from two distinct networks.

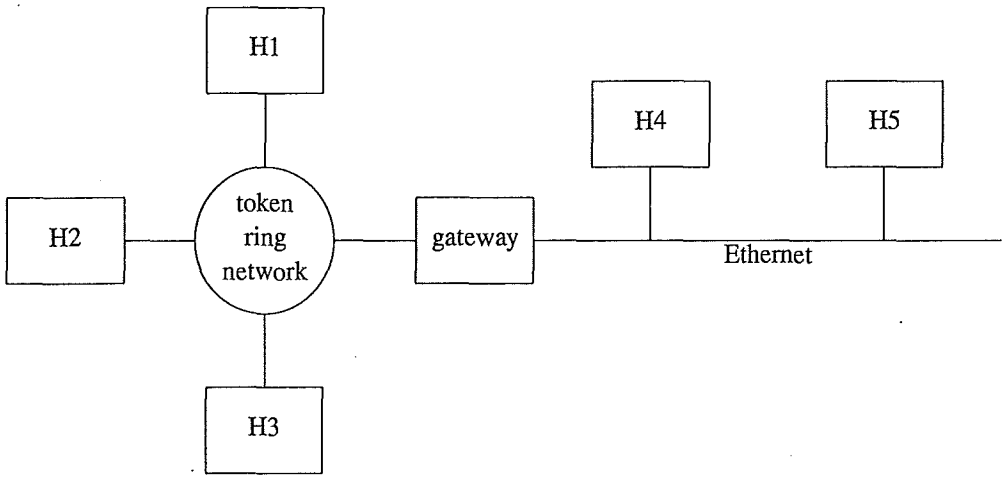


Figure 4.1 Internet example with two networks joined by a gateway.

There are three hosts on the token ring, H1, H2, and H3, and two hosts on the Ethernet, H4 and H5. There is a gateway between the two physical networks, and it must contain an interface card to attach to the token ring network and another interface card to attach to the Ethernet. It must also determine when information arriving on the Ethernet is destined for a host on the token ring (and vice versa), and handle it appropriately.

There are more ways to connect networks together. The term we use to describe the interconnection depends on the layer in the OSI model at which the connection takes place.

- *Repeaters* operate at the physical layer (layer 1) and typically just copy electrical signals (including noise) from one segment of a network to the next. Repeaters are often used with Ethernets, for example, to connect two cable segments together to form a single network.

- *Bridges* often operate at the data-link layer (layer 2) and they copy frames from one network to the next. Bridges often contain logic so that they only copy a subset of the frames they receive.
- *Routers* operate at the network layer (layer 3). The term router implies that this entity not only moves information (packets) from one network to another, but it can also make decisions about what route the information should take. (We talk more about routing later in this chapter.)
- *Gateway* is a generic term that refers to an entity used to interconnect two or more networks. In the TCP/IP community, for example, the term gateway refers to a network level router. The term gateway is sometimes used to describe software that performs specific conversions at layers above the network layer. For example, there are various mail gateways that convert electronic mail from one format to another. We mention an application-gateway in Section 5.6 that converts file transfer requests between the OSI FTAM protocol and the TCP/IP FTP protocol. We'll use the term gateway to describe a network level router, unless stated otherwise.

Repeaters are usually hardware devices, while bridges and routers can be implemented in either hardware or software. A router (gateway) is usually a dedicated system that only does this function. 4.2BSD, however, was the first general-purpose system that could also operate as a gateway.

A host is said to be *multihomed* if it has more than one network interface. For example, a host with an Ethernet interface and a token ring interface would be multihomed.

Looking at the different levels of abstraction, we go from a user sitting at a terminal to an internet.

- *Users* login to a host computer.
- *Host* computers are connected to a network.
- *Networks* are connected together to form an internet.

An internet of computer networks is similar in principle to the international long distance telephone service that is available today. Telephones are connected to a local phone company, which in turn is connected into a national long distance network, which is then connected into an international network. When you direct dial an international call, this "telephone internet" hides all the details and connects all the telephones into a coordinated unit. This is the goal of an internet of computer networks.

OSI Model, Protocols, and Layering

The computers in a network use well-defined *protocols* to communicate. A protocol is a set of rules and conventions between the communicating participants. Since these protocols can be complex, they are designed in layers, to make their implementation more

manageable. Figure 4.2 shows the OSI model that was introduced in Section 1.3.

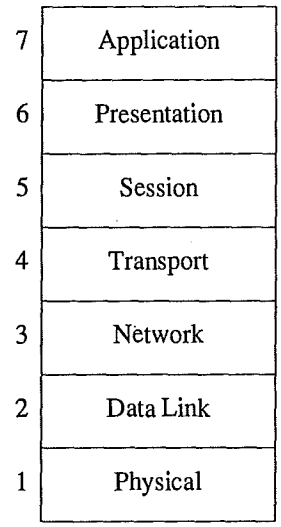


Figure 4.2 OSI model.

This model, developed between 1977 and 1984, is a guide, not a specification. It provides a framework in which standards can be developed for the services and protocols at each layer. Indeed, the networks that we consider in this text (TCP/IP, XNS, and SNA) were developed before the OSI model. We will compare the layers for these actual networks against the OSI model, but realize that no network is implemented exactly as the OSI model shows.

One advantage of layering is to provide well-defined interfaces between the layers, so that a change in one layer doesn't affect an adjacent layer. It is important to understand that protocols exist at each layer. A *protocol suite* is a collection of protocols from more than one layer that forms the basis of a useful network. This collection is also referred to as a *protocol family*. The protocol suites that we describe in Chapter 5 are

- the TCP/IP protocol suite (the DARPA Internet protocols),
- Xerox Network Systems (Xerox NS or XNS),
- IBM's Systems Network Architecture (SNA),
- IBM's NetBIOS,
- the OSI protocols,
- UUCP.

Each of these protocol suites define different protocols at different layers, as we see in the next chapter.

Recall from Section 1.5 that we simplified the OSI model into a 4-layer model that we use in the text. Figure 4.3 shows this model for two systems that are connected with a network.

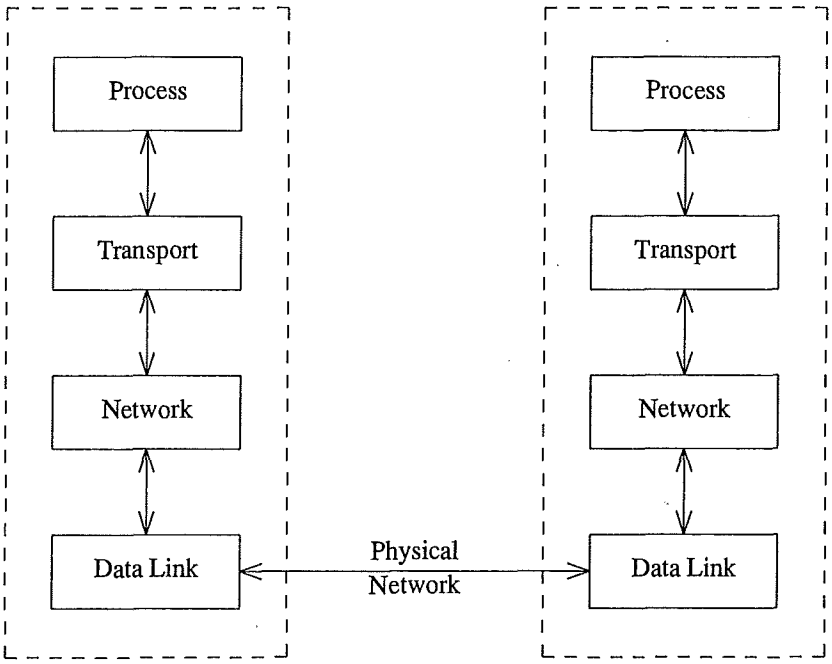


Figure 4.3 Simplified 4-layer model connecting two systems.

The layers that define a protocol suite are the two boxes called the *transport layer* and the *network layer*. The multiple layers that define the network and hardware characteristics (Ethernet, token ring, etc.) are grouped together into our *data-link* layer. Application programs exist at the *process layer*. It is the interface between the transport layer and the process layer that we describe in Chapters 6 and 7.

Let's jump ahead to provide a concrete example for layering and some of the other concepts described in this chapter. Figure 4.4 shows the layering used by the TCP/IP protocol suite.

Consider one specific user process whose protocol is defined by the TCP/IP protocol suite, TFTP, the Trivial File Transfer Protocol. This application allows a user on one system to send and receive files to and from another system. TFTP uses UDP (User Datagram Protocol), which in turn uses IP (Internet Protocol), to exchange data with the other host.

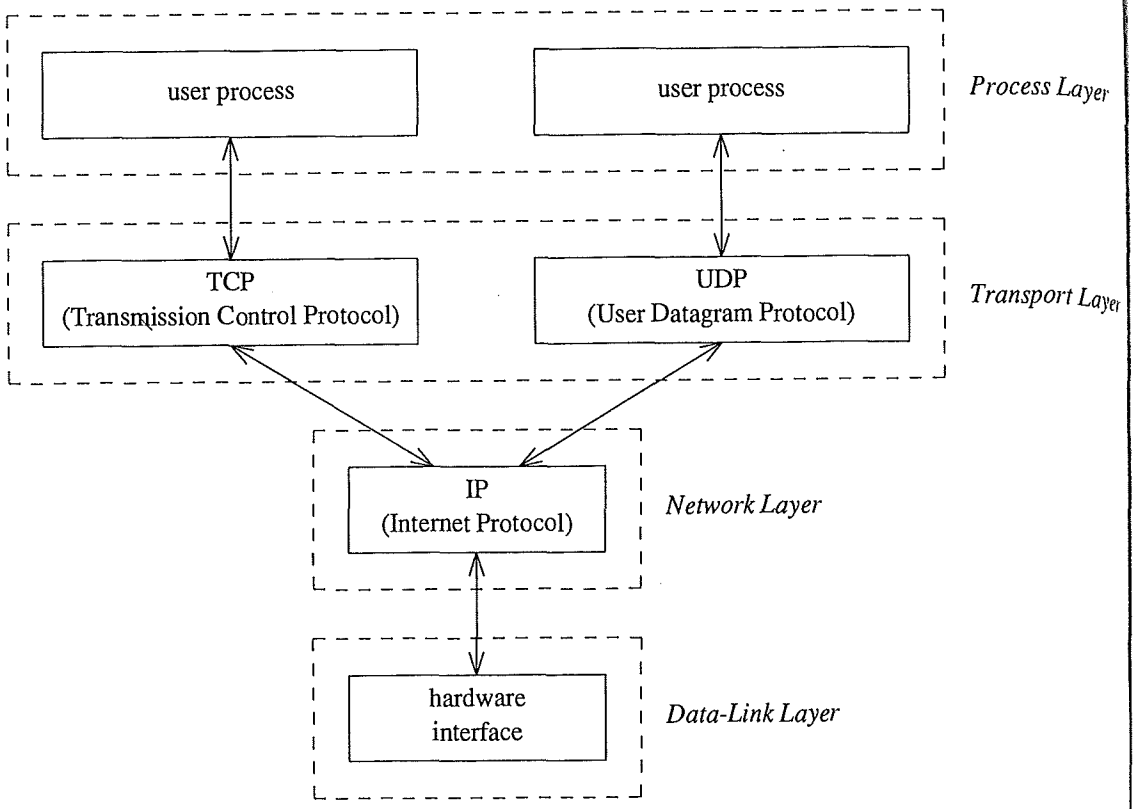


Figure 4.4 TCP/IP protocol suite using 4-layer model.

If we consider two hosts connected with an Ethernet, Figure 4.5 shows the four layers.

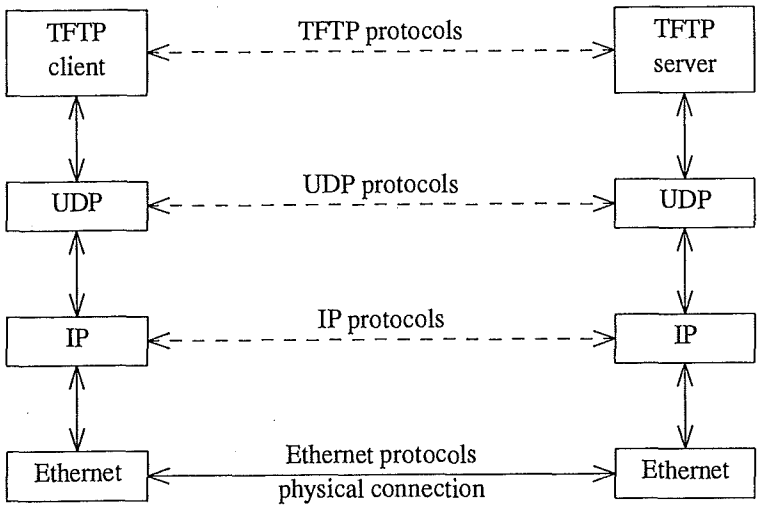


Figure 4.5 4-layer model for TFTP over UDP/IP on an Ethernet.

We show the protocols between the top three boxes with dashed lines to indicate that the boxes at these three layers communicate "virtually" with each other using the indicated protocols. Realize that only the boxes at the lowest layer physically communicate with each other. We call these horizontal lines between layers at the same level *peer-to-peer protocols*. The actual flow of data is from the TFTP box on one side, down to the UDP box, down to the IP box, down to the Ethernet box, then across the physical link to the other side and up.

Another goal of layering, besides making the protocol suite easier to understand, is to allow us to replace the contents of a layer with something else that provides the same functionality. With the example shown in Figure 4.5, if we are able to connect the two computer systems with a token ring network, we would have the layers shown in Figure 4.6.

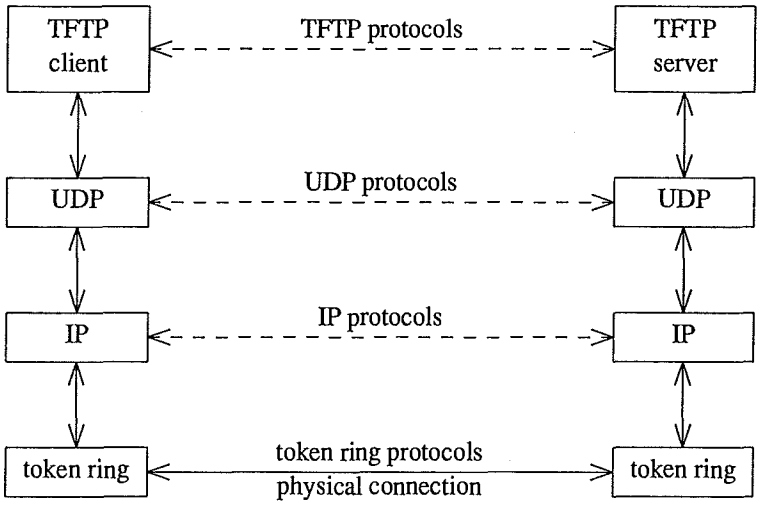


Figure 4.6 4-layer model for TFTP over UDP/IP on a token ring.

The TFTP protocols do not have to change at all between this network and the previous one that used an Ethernet. Similarly, the UDP/IP protocols do not have to change either.

We can take this example one step further and replace the middle two boxes that use the UDP/IP protocols from the TCP/IP protocol suite, with boxes that use the PEX/IDP protocols from the Xerox NS protocol suite. If this were done using an Ethernet connection between the two systems, we would have the layers shown in Figure 4.7. Again, the TFTP protocols do not have to change at all between the version that uses the UDP/IP protocols and the version that uses the Xerox PEX/IDP protocols.

In Chapter 12 we develop a complete implementation of a TFTP client and server and implement it using different transport protocols.

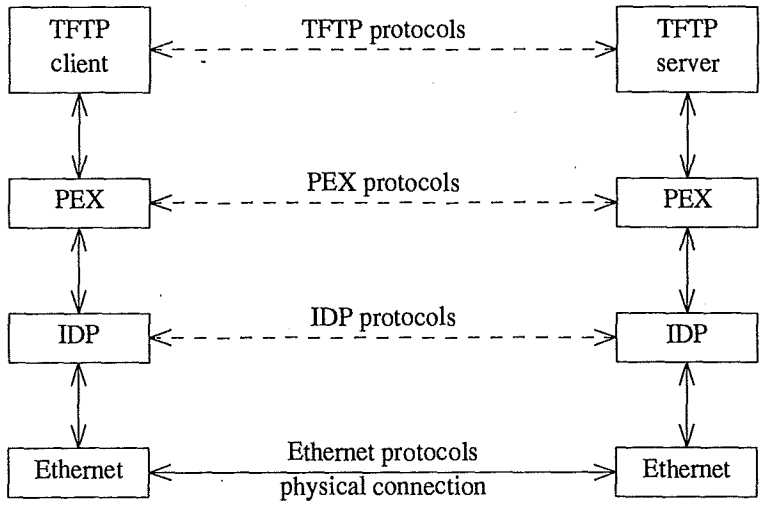


Figure 4.7 4-layer model for TFTP over PEX/IDP on an Ethernet.

Bytes and Octets

A term that appears in the networking literature is *octet*, which means an 8-bit quantity of data. We will consistently use the term *byte* instead of octet. The term octet arose because some computer systems, notably the DEC-10 series and the Control Data Cyber series, don't use 8-bit bytes. Fortunately, however, virtually all modern computers use 8-bit bytes, so we use the term byte to mean an 8-bit byte.

Network Byte Order

Unfortunately, not all computers store the bytes that comprise a multibyte value in the same order. While the problems with systems that don't use 8-bit bytes are slowly disappearing, as these older systems fade into history, the problems with byte ordering will be with us for a long time, as there is no clear standard.

Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store this value: with the low-order byte at the starting address, known as *little endian*, or with the high-order byte at the starting address, known as *big endian*. The first case is shown in Figure 4.8.

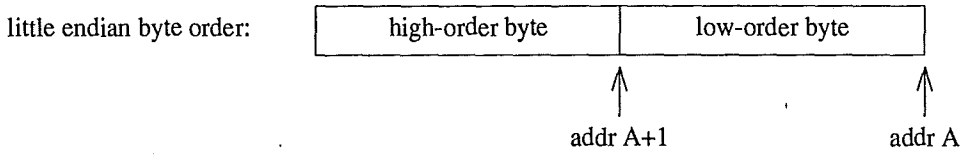


Figure 4.8 Little endian byte order for a 16-bit quantity.

Here we are looking at increasing memory addresses going from *right to left.* The reasoning behind this byte ordering is that a lower address implies a lower order byte.

The big endian format can be pictured as in Figure 4.9.

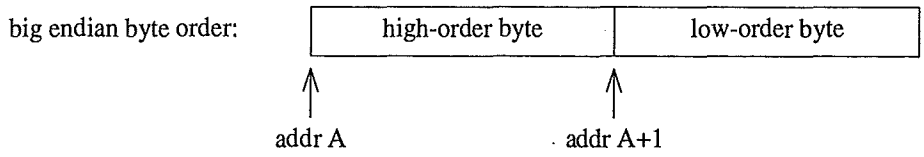


Figure 4.9 Big endian byte order for a 16-bit quantity.

Here we are looking at increasing memory addresses going from *left to right.*

The byte ordering used by some current computer systems is shown in Figure 4.10.

big endian:	IBM 370, Motorola 68000, Pyramid
little endian:	Intel 80x86 (IBM PC), DEC VAX, DEC PDP-11

Figure 4.10 Byte ordering used by different systems.

The byte ordering with 32-bit integers is even worse, as some systems swap the two 16-bit pieces of the 32-bit integer.

The solution to this problem is for a network protocol to specify its *network byte order.* The TCP/IP, XNS, and SNA protocols all use the big endian format for the 16-bit integers and 32-bit integers that they maintain in the protocol headers. (Fortunately the protocols maintain only integer fields, as the differences in the internal formats for floating point data are even worse.) The protocol has no control over the format of the data that the applications transfer across the network—the protocol only specifies the format for the fields that it maintains. We will be concerned with the network byte ordering of multibyte integer fields in Chapter 6 when we discuss the interface between a user process and the networking software. We also return to the problem of application data format in Chapter 18 when we discuss remote procedure calls.

Encapsulation

Let's again consider the TFTP application, using the UDP/IP protocols, between two systems connected with an Ethernet. If there are 400 bytes for the TFTP client process to transfer to the TFTP server process, the TFTP client process adds 4 bytes of control information to the beginning of the data buffer, before passing the data down one layer to the UDP layer. This addition of control information to data is called *encapsulation* and is shown in Figure 4.11.

The UDP layer does not interpret the 4-byte TFTP header at all. The task of the UDP layer is to transfer 404 bytes of data to the other UDP layer. The UDP layer then prepends its own 8-byte header and passes the 412-byte buffer to the IP layer. The IP

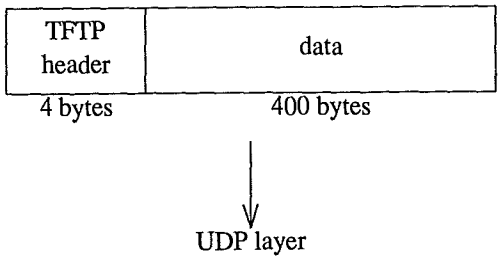


Figure 4.11 Encapsulation of user data by TFTP.

layer prepends its 20-byte header and passes the 432-byte buffer to the data-link layer for the Ethernet. At this layer a 14-byte header and a 4-byte trailer are added to the buffer of information. Figure 4.12 shows this encapsulation by each layer.

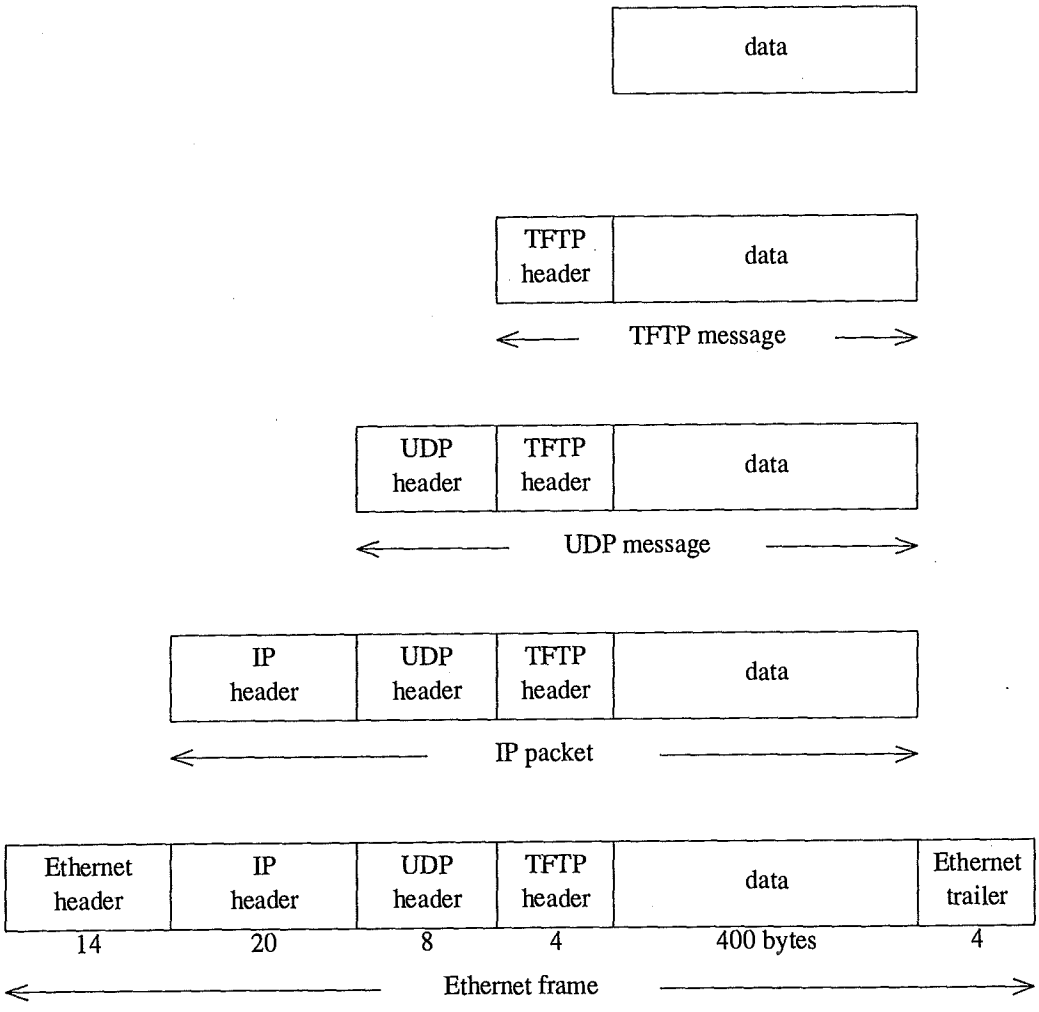


Figure 4.12 Encapsulation of TFTP over UDP/IP on an Ethernet.

The final diagram shows the Ethernet frame that is physically transmitted across the Ethernet, along with the sizes of each of the headers and trailers, in bytes.

We define the units of information that are passed along a network according to the layer at which the transfer is taking place. This is shown in Figure 4.13.

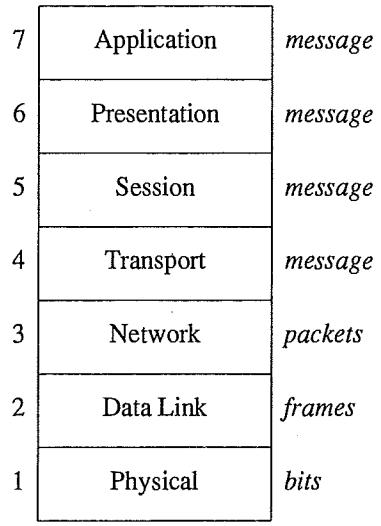


Figure 4.13 OSI model with units of information exchanged at each layer.

We call the unit of interchange at the network layer *packets*. At the data-link layer we call them *frames*, and at the lowest layer, the physical layer, *bits* are exchanged.

Multiplexing and Demultiplexing

Multiplexing means to combine many into one, and we have examples of this happening at most layers in a network. Consider the multihomed, multiuser computer system shown in Figure 4.14. This figure shows applications using both UDP and TCP from the TCP/IP protocol suite along with PEX and SPP from the Xerox NS protocol suite. Both of these protocol suites can use a single Ethernet. There are many points to be made from Figure 4.14.

- A user process can use different protocols at the same time. For example, process B is using both UDP and TCP at the same time. Similarly, process D is using TCP and PEX.
- More than one user process at a time can be using any of the user-accessible protocols (UDP, TCP, PEX, or SPP). This requires that a given protocol box, the UDP box for example, must identify the user process (A or B) that is sending data when the UDP software passes this data down to the IP layer. This is one example of *multiplexing*. Conversely, when the UDP software receives data from the IP layer, it must be able to identify the user process to receive the data. This is *demultiplexing*. For these examples, both UDP and TCP have a 16-bit port number that identifies the user process. We discuss this field in the next chapter.

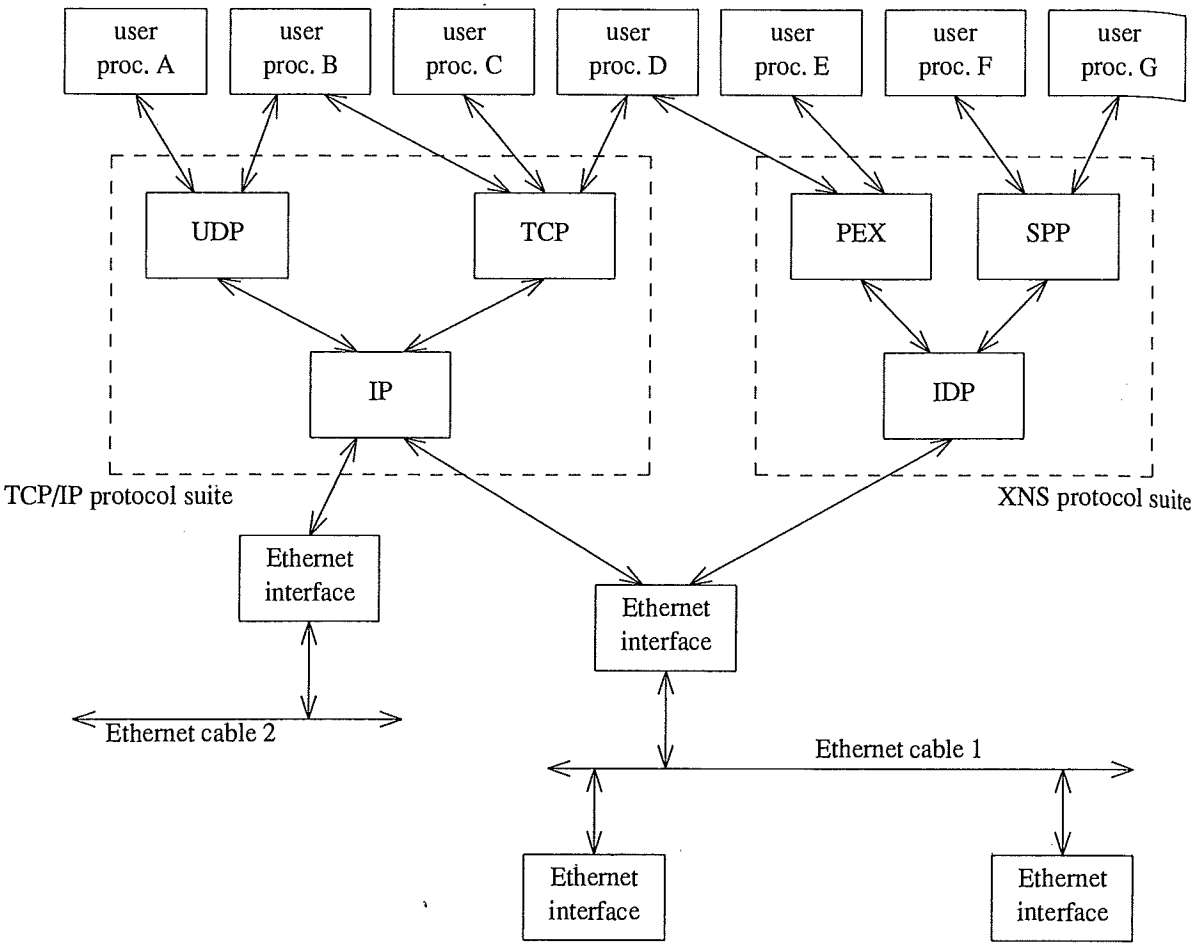


Figure 4.14 Multiplexing and demultiplexing.

- The IP module must determine if data that it receives from the Ethernet interface is for the UDP module or the TCP module. The IP header contains an 8-bit protocol field that is used for this purpose.
- We show the computer system with two Ethernet interfaces. This system is a gateway between the two networks. The IP module must determine which interface is to be used for packets from either the UDP module or the TCP module.
- The software (the device driver) for the interface connected to Ethernet cable 1 must determine if a received Ethernet frame is for the IP module or the IDP module. There is a 16-bit field in the Ethernet header that identifies the frame type. This allows multiple protocol suites, such as the TCP/IP suite and the XNS suite, to share the same Ethernet.

- An Ethernet interface must determine if a frame that is being transmitted across the physical cable is for it, or for another interface. This determination is done by the Ethernet hardware. Each Ethernet frame contains a 6-byte address in its header that identifies the destination interface. Ethernet addresses are usually supplied by the hardware vendor so that every interface has a unique 6-byte address.

Packet Switching

Communication networks can be divided into two basic types: *circuit-switched* and *packet-switched*. The classic example of a circuit-switched network is the public telephone system. When you place a telephone call, either local or long distance, a dedicated circuit is established for you by the telephone switching offices, from your telephone to the other telephone. Every telephone is directly connected to a local office, typically within a few miles of the phone. The local offices are then connected to toll centers, and these toll centers are connected together through sectional centers and regional centers. Once this circuit is established, the only delay involved in the communication is the time required for the propagation of the electromagnetic signal through all the wires and switches. While it might be hard to obtain a circuit sometimes (such as calling long distance on Christmas day), once the circuit is established you are guaranteed exclusive access to it.

A leased telephone line, which is common for WANs, is a special case of a circuit-switched network. But there is no setup required to establish a dedicated circuit between the two telephones.

An internet, on the other hand, typically uses packet-switching techniques. Instead of trying to establish a dedicated communication line between one computer and another, the computers share communication links. Instead, the information is divided into pieces and each piece is transmitted on its own through the connection of networks. These pieces are called *packets*.

A packet is the smallest unit that can be transferred through the networks by itself. A packet must contain the address of its final destination, so that it can be sent on its way through the internet. Most protocols also specify that a packet contain the sender's address, too. (We describe these addresses later in the chapter.) Recall from earlier in this chapter that we defined the unit of information that is exchanged at the network layer is a packet. Indeed, it is the network layer that is involved in transferring packets around the networks.

With a circuit-switched network, we are guaranteed that once a circuit is established, we can use the full capacity of the circuit. With a packet-switched network, however, we are sharing the communication bandwidth with other computers.

Packet switching on a local area network refers to multiple computers sharing a single communications channel, such as an Ethernet or a token ring. A dedicated link is not used between each computer. All the computers on the LAN share the available capacity of the network.

A WAN can also use packet switching. We'll show an example of this, after discussing gateways.

Consider the internet shown in Figure 4.15.

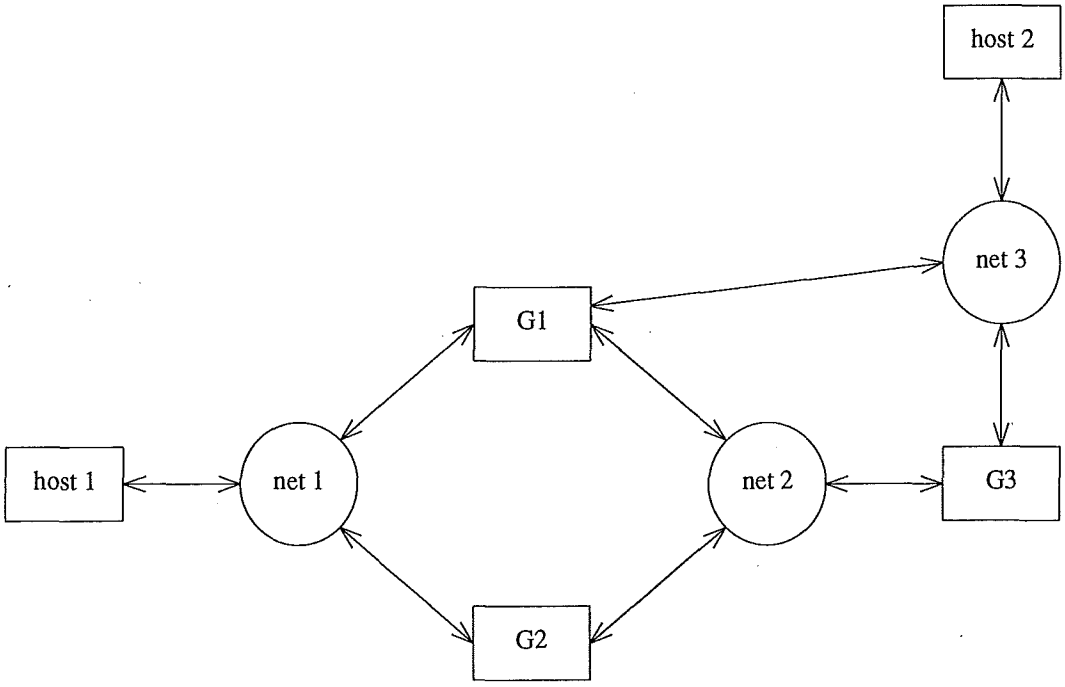


Figure 4.15 Internet example for routing.

For a packet to go from host 1 to host 2, there are four possible acyclic paths.

- host 1, net 1, G1, net 3, host 2;
- host 1, net 1, G1, net 2, G3, net 3, host 2;
- host 1, net 1, G2, net 2, G3, net 3, host 2;
- host 1, net 1, G2, net 2, G1, net 3, host 2.

Each of these paths is called a *route*. Routing decisions are usually made at the network layer in the OSI model. We'll consider routing in more detail later in this chapter.

In a packet-switched network, there is no guarantee how long it takes a packet to go from one host to another. The time taken for each packet depends on the route chosen for that packet, along with the volume of data being transferred along that route.

Gateways

Earlier we defined a *gateway* as a system that interconnects two or more networks. The function of a gateway is to pass along information from one network to another.

The layer at which a gateway operates depends on the type of translation and forwarding done by the gateway. For example, if we have an internet of networks that all use the TCP/IP protocols, no translation of protocols is required. Instead, each gateway only needs to forward packets from one network to another. The IP layer is responsible for this forwarding of packets. An example of this is shown in Figure 4.16.

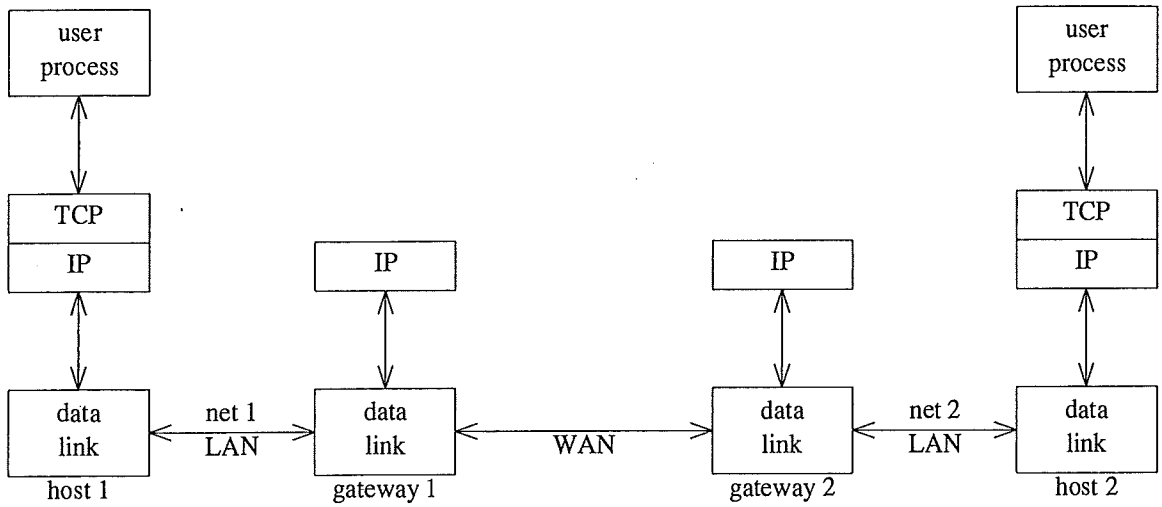


Figure 4.16 Gateway example.

Each IP packet contains enough information (i.e., its final destination address) for it to be routed through the TCP/IP internet by itself.

Things become more complicated, if we want a gateway between networks that use different protocol suites. The conversion is often done in layers above the network layer.

Notice in Figure 4.16 that the TCP layer is not needed on the intermediate gateways. This is why we distinguished between the transport layer and the network layer, in Figures 4.3 and 4.4. You might think we could combine them, since they often appear as a single entity within the operating system, to the application programmer. Our reason for separating them is because of the different roles they each play in an internet. The transport layer is the lowest layer that provides the virtual connection between the two hosts on which the user processes are running.

To understand packet switching in a WAN, let's look at Figure 4.16 again. Assume the two LANs are Ethernets and assume that the two gateways are separated by thousands of miles but are connected with a leased phone line. Even though this leased

phone line is a circuit-switched connection, we still consider this a packet-switched network. The reason is there are usually many hosts that share the Ethernets connecting each host and its gateway. (The number of hosts on an internet typically exceeds the number of gateways on the internet by a factor of 10 or more. For example, the TCP/IP Internet is currently estimated to have over 150,000 hosts and 850 networks.) All the hosts on one of these Ethernets compete with the other hosts on that same Ethernet for the single link between the two gateways. Nevertheless, this arrangement is more reasonable than a leased phone line between every host on network 1 and every host on network 2.

Fragmentation and Reassembly

Most network layers have a maximum packet size that they can handle, based on the characteristics of the data-link layer that they are sending its packets through. This is called the network's *maximum transmission unit* or *MTU*. (This term can be applied to the maximum size of a protocol message exchanged by any two layers in a protocol suite, not just the network layer.) For example, part of the Ethernet standard is that the data portion of a frame (the data between, but not including, the 14-byte Ethernet header and the 4-byte Ethernet trailer) cannot exceed 1500 bytes. The maximum for a token ring network is typically 4464 bytes, while some networks have small MTUs. Consider the example shown in Figure 4.16 and assume that the two LANs are Ethernets. But, if the WAN between the two gateways uses a data link with a maximum packet size of 128 bytes, then something has to be done when IP packets are sent from host 1 to host 2.

Fragmentation is the breaking up of a data stream into smaller pieces. Some networks use the term *segmentation* instead of fragmentation. The reverse of fragmentation is *reassembly*. Using the layering concept that we've already talked about, fragmentation should happen at the lowest layer possible, so that upper layers need not be concerned with it. For example, using the gateway example in Figure 4.16, we would like fragmentation to take place at the IP layer, without the TCP layer even knowing that it was taking place. Indeed, with the TCP/IP protocols, it is the IP layer that handles fragmentation and reassembly, as required by the different data-link layers that are used by TCP/IP.

Modes of Service

There are several parameters that describe the type of communication service provided between two peer entities at any layer of the OSI model. In this text we are interested in the service provided to an application program, typically by the transport layer. These parameters are

- connection-oriented (virtual circuit) or connectionless,
- sequencing,
- error control,
- flow control,

- byte stream or messages,
- full-duplex or half-duplex.

A *connection-oriented* service requires that the two application programs establish a logical connection with each other before communication can take place. There is some overhead involved in establishing this connection. We also use the term *virtual circuit* to describe this service, since it appears to the application programs that they have a dedicated circuit between them, even though the actual data flow usually takes place using a packet-switched network. A connection-oriented service is often used when more than one message is to be exchanged between the two peer entities. A connection-oriented exchange involves three steps:

- connection establishment,
- data transfer (can be lengthy),
- connection termination.

The converse of a connection-oriented service is a *connectionless* service, also called a *datagram* service. In this type of service, messages called datagrams are transmitted from one system to the other. Since each message is transmitted independently, each must contain all the information required for its delivery. In the TCP/IP protocol suite, TCP provides a connection-oriented virtual circuit, while UDP provides a connectionless datagram facility.

Sequencing describes the property that the data is received by the receiver in the same order as it is transmitted by the sender. In a packet-switched network, it is possible for two consecutive packets to take different routes from the source computer to the destination computer, and thus arrive at their destination in a different order from the order in which they were sent.

Error control guarantees that error-free data is received by the application programs. There are two conditions that can generate errors: the data gets corrupted (modified during transmission), or the data gets lost. A technique to detect data corruption is for the sender to include a checksum so the receiver can verify, with a high probability, that the data does not get modified. If the data does get corrupted, the receiver has to ask the sender to retransmit the data. Therefore, checksums are usually combined with a technique called positive acknowledgment—the receiver notifies the sender each time a data message is received, either correctly or with errors. If the data was received correctly the sender can discard it, otherwise the data must be retransmitted. To handle the loss of data somewhere in the network requires that the sender start a timer after it has sent a data message, and if the timer expires (called a timeout) the sender must retransmit the data. Note that when positive acknowledgments and timeouts are being used, it is possible not only for data to get lost but for acknowledgments to be lost as well. If this happens, the original sender will retransmit the data, causing the other end to receive the same data twice. This requires the receiver to perform duplicate detection—determine when data has already been received and ignore (but acknowledge) the duplicate message.

Flow control assures that the sender does not overwhelm the receiver by sending data at a rate faster than the receiver can process the data. This is also called *pacing*. If flow control is not provided, it is possible for the receiver to lose data because of a lack of resources. Both TCP and SNA LU 6.2 provide sequencing, error control, and flow control. UDP, however, does not provide any of these services.

A *byte-stream* service does not provide any record boundaries to the data stream. The converse of this feature is a message-oriented service that preserves the sender's message boundaries for the receiver. These features were described in Section 3.6. TCP is a byte-stream protocol, while UDP and the XNS SPP (Sequenced Packet Protocol) provide message boundaries.

A *full-duplex* connection allows data to be transferred in both directions at the same time between the two peer entities. Some protocols are *half-duplex* and only allow one side to transfer at a time. TCP is full-duplex, while SNA LU 6.2 is a half-duplex protocol.

Many of these parameters occur together. For example, sequencing and error control are usually provided together and the protocol is then termed *reliable*. It is also unusual to find a reliable protocol that does not provide flow control. It is unusual to have a connectionless protocol that provides sequencing, since the datagrams in such a protocol are usually unrelated to previous or future datagrams. We see an example of sequencing, positive acknowledgment, and retransmission after timeout in Chapter 12, when we develop an implementation of the Trivial File Transfer Protocol.

Tanenbaum [1989] provides an excellent analogy between virtual circuits and datagrams: A telephone call is similar to a virtual circuit. You establish a connection with the other party, exchange data for a while, then close the connection. The postal service resembles a datagram delivery service. Every letter contains the complete address of the destination. Every letter that you mail to the same address can follow a different route to the destination. (No doubt everyone has mailed two letters to the same destination, one or two days apart, only to have them arrive out of order.) There is no guarantee that a letter arrives at the destination. To receive an acknowledgment of delivery you must add extra processing to the normal mailing of a letter (i.e., pay extra for a "return receipt" if you're using the U.S. Postal Service). If you are expecting a response from your letter, you must also add extra processing—wait for some period and if a response is not received (timeout), retransmit the letter.

End-to-End versus Hop-by-Hop

When describing protocol features such as flow control, error control, and the like, we must differentiate between those features that are done on an *end-to-end* basis and those that are done on a *hop-by-hop* basis. End-to-end flow control, for example, regulates the flow of data between the two user processes that are exchanging data. Hop-by-hop flow control, on the other hand, worries about the flow of data at each hop of the communication link (i.e., all the gateways between the two end systems). As we'll see in the next chapter, TCP provides end-to-end flow control, while IP provides an elementary type of hop-by-hop flow control.

As another example, error control is usually handled on a hop-by-hop basis by the data-link layer. An Ethernet frame, a token ring frame, or an SDLC frame, all contain a checksum that detects any modifications to the data as it travels across the physical link. This is a hop-by-hop form of error detection. Some protocols also provide some form of end-to-end error control also. The TCP layer in the TCP/IP suite provides this—the source TCP layer calculates and stores a checksum in the TCP message header which is then checked by the receiving TCP layer.

Looking at Figure 4.16 again, we see that any feature that is to be implemented in the TCP/IP protocol suite on an end-to-end basis, has to be handled by the TCP layer, or the user process above TCP. Similarly, any hop-by-hop feature must be handled by the IP layer or the data-link layer.

Buffering and Out-of-Band Data

Consider a byte-stream service. The user process can read or write any number of bytes at a time, since there are no message boundaries. What typically happens is that the byte-stream service layer (TCP, for example) buffers this data internally and then passes it to the next lower layer for transmission to the other end. Buffering can also take place if the protocol does flow control (as most byte-stream service protocols do) since the sending side can be generating data faster than the receiving end can process the data. Also, a reliable protocol can be buffering data on the sending side while it is waiting for an acknowledgment of previously sent data from the other end. Note that buffering can be taking place on both ends, as the receiving byte-stream service can be receiving data from the network faster than the user process is reading the data. Figure 4.17 shows this buffering on both hosts.

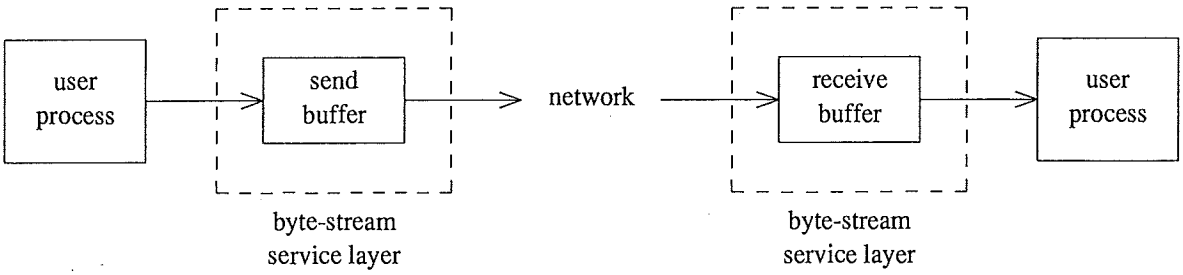


Figure 4.17 Normal buffering that takes place with a byte-stream service.

Sometimes a user process wants to disable this buffering. For instance, consider a terminal emulation program. The process that is reading the keystrokes wants to send each keystroke to the other end, one byte at a time, since the other end might be operating on each character as it is entered. This is how full screen editors under Unix operate, for example. Similarly, the process that is sending terminal output also wants to force the data to be transferred, perhaps at the end of every line. What we would like is the ability for a user process to tell its byte-stream service layer (e.g., TCP) to flush any data that is has buffered and send it to the other end. TCP, for example, supports this option, which it calls a *push*.

This type of buffering sounds like the type done by the standard I/O library, but network buffering quickly becomes more complex. The complicating factors are that some network applications are interactive (e.g., remote echoing in a remote login application), which makes us want to buffer as little as possible, while performance considerations make us want to buffer as much as possible. Also, congestion can quickly become a problem if we send many small packets through a busy network, since there is a fixed overhead for every packet, no matter how much data it contains. The techniques used by the TCP implementation in 4.3BSD are covered in Section 12.7 of Leffler et al. [1989]. (We consider the buffering techniques of the 4.3BSD TCP implementation in more detail in Section 6.11 when we describe the `TCP_NODELAY` socket option.)

Similarly, when a user at a terminal presses the attention key, it needs to be operated on as soon as possible. The Unix interrupt key (typically the Delete key or Control-C) is one example of this, as are the terminal flow control characters (typically Control-S and Control-Q). This type of information is termed *out-of-band data* or *expedited data*. With out-of-band data we want the byte-stream service layer on the sending side to send this data before any other data that it has buffered. Similarly we want the receiving end to pass this data to its user process ahead of any data that it might have buffered. Some method is also desired for the receiving byte-stream service layer to notify the user process asynchronously that out-of-band data has arrived. As you might guess, a Unix signal can be used for this notification. The term *out of band* is used because it appears to the user processes that a separate communication channel is being used for this data, in addition to the normal data channel (band). This is shown in Figure 4.18.

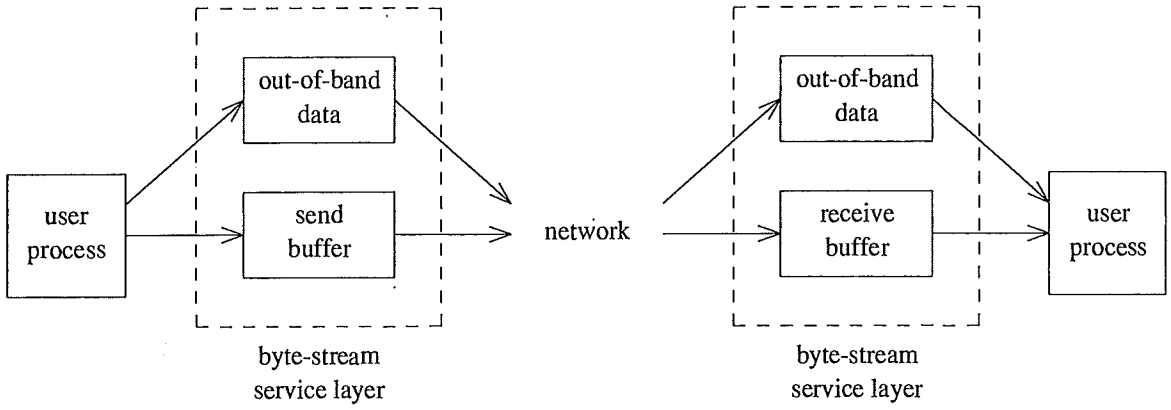


Figure 4.18 In-band data and out-of-band data.

Some byte-stream layers provide out-of-band data and some don't. Those that do provide it allow differing amounts. For example, SPP provides a single byte of out-of-band data, while TCP allows an unlimited amount. We return to both of these features, buffer flushing and out-of-band data, in the next three chapters when we examine each specific protocol family and then examine the user process interface to the transport layer.

Services and Protocols

It is necessary to differentiate between *services* and *protocols* in a layered model such as the OSI model. A layer provides one or more services to the layer above it. As mentioned earlier, this text is mainly concerned with the services provided by the transport layer to an application. Earlier we defined a protocol as a set of rules and conventions that the communicating participants must follow, to exchange information.

In many of the protocol suites that preceded the OSI model, the services provided by the different layers are not distinguished from the protocols that implement these services. Most of the OSI protocols, however, separate the two, as we'll see in Section 5.6. The advantage in separating the service from the protocol is that multiple protocols can be developed over time to provide a given service.

Chapter 5 is concerned mainly with protocols, although there will be some implied service features for those protocols that don't differentiate between services and protocols. Chapters 6 and 7, however, deal with the services provided by the transport layer to a user process—Berkeley sockets and System V TLI.

Addresses

Recall that the end points for communication are two user processes, one on each system. With an internet, the two systems can be located on different networks, connected by one or more gateways. This requires three levels of addressing.

- A particular network must be specified. As we've seen earlier in this chapter, a given host can be connected to more than one network.
- Each host on a network must have a unique address.
- Each process on a host must have a unique identifier on that host.

Typically a host address consists of a network ID and a host ID on that network. The identification of the user process on a host is often done with an integer value assigned by the protocol. For example, the TCP/IP protocol suite uses a 32-bit integer to specify a combined network ID and host ID, and both TCP and UDP use 16-bit port numbers to identify specific user processes. The XNS protocol suite uses a 32-bit network ID, a 48-bit host ID, and a 16-bit port number. In the next chapter we'll examine the specific address formats used by each protocol suite in more detail.

Most protocol suites define a set of *well-known addresses* that are used to identify the well-known services that a host can provide. For example, most TCP/IP implementations provide a file transfer server named FTP that a client can contact to transfer a file. The 16-bit integer port for FTP is 21. In Chapter 8 we discuss how a service name such as FTP is mapped into a protocol-specific address, since it differs between protocol suites and between implementations.

Broadcast and Multicast

Consider a single network without any gateways. A *unicast* message is sent by a host to one specific host on the network. A *broadcast* message is one that is transmitted by a host to every host on the network. A *multicast* message is one that is transmitted by a host to a specified group of hosts on the network. Some types of local area network technology support broadcast messages at the data-link layer—Ethernet and token ring, for example. A special destination address specifies a broadcast message. An Ethernet destination address of all 1-bits, for example, specifies a broadcast. Other forms of network connections, such as point-to-point phone lines, require some form of software implementation of broadcast.

A multicast message implies that some subset of the host systems belong to a multicast group. Some forms of network hardware provide support for multicast addresses, but this is more product specific than support for broadcast messages. For example, some (but not all) Ethernet interfaces allow the host to specify those multicast groups to which it belongs to.

When a network is connected to another network through a gateway, should broadcast messages generated on the local network be passed along by the gateway to the next network? In normal practice, broadcast messages are not forwarded.

We will see examples of broadcast messages in the next chapter. Multicasting is not as widely supported as broadcasting.

Routing

The topic of network routing is complex and could consume an entire text of its own. Here we'll provide a brief overview. Note that the applications programmer typically has little control over routing.

When a node, either a host or a gateway, has a packet to send to some other host, there are four possible cases.

1. The destination host is either directly connected to the node (with a leased phone line, for example), or the destination host and the node are on the same network (an Ethernet, for example). In either case the node can send the packet directly to the destination host—no routing decision has to be made.
2. The arriving packet can contain the address of the next node to send the packet to. This is called *source routing* since the list of addresses for the packet to follow is established by the source host. No decisions have to be made by the current node about where to send the received packet. This technique is used by the IBM token ring [Dixon and Pitt 1988]. Additionally, the TCP/IP protocol suite has an option that allows source routing to be used, however, this is typically used only by network administrators for testing. See Section 7.8.2 of Comer [1988].

3. The current node can already know the route that packets from this source are to follow to reach the specified destination. There are some packet-switched networks in which every packet from a given source to a given destination follow the same route. In this case there are no decisions to be made by the current node. The commercial TYMNET network, for example, uses this technique, as does the IBM SNA/LEN product [Baratz et al. 1985].
4. None of the three cases above apply. A routing decision must be made by the current node. This, for example, is the normal mode of operation for a gateway on the TCP/IP Internet. Source routing is not typically used and two consecutive datagrams from a given host to the same destination can follow different paths.

Note that some routing decision has to be made for cases 2, 3, and 4 above. In cases 2 and 3 the decision is made before the node receives the packet, so the node itself doesn't have to make a decision. Nevertheless, someone had to make a routing decision for this packet. In case 4 the routing decision is made on-the-fly by the node, which is typically a gateway.

When routing decisions have to be made, there are four different types of algorithms that can be used.

- static routing,
- isolated dynamic routing,
- centralized dynamic routing,
- distributed dynamic routing.

The routing algorithm can be applied by the source host (case 2), by a central authority (case 3), or by the gateway itself (case 4).

Static routing chooses a route based on precomputed information. There is no consideration of the current network dynamics—the load between gateways, for example. The Unix `pathalias` program generates static routing tables that are used by the Unix mailers. Sites typically run this program at some interval (every night or every week) to compute the routes that UUCP mail will follow.

Dynamic routing takes into account the current load on the network. The first dynamic technique, isolated dynamic routing, bases its decisions on local information only. Typically this is the amount of data that the gateway currently has to send, or has recently sent, out each of its interfaces. This technique is rarely used.

Centralized dynamic routing has a central authority that makes all routing decisions based on current network information. The commercial TYMNET network, for example, uses this technique. Each gateway regularly reports its traffic information to a central node and this central node uses all this information to choose routes. Since a source host must contact the central node each time a routing decision is made, realistically this technique is only applicable to networks that use virtual circuits at the network layer—all packets from a given host to a given destination follow the same path. This way the central node only has to be contacted when a new virtual circuit is established.

Distributed dynamic routing uses a mixture of global and local information to make routing decisions. In order for a gateway to obtain global information, the gateways must have some way of exchanging this information. The TCP/IP Internet, for example, uses distributed dynamic routing. Narten [1989] discusses the history of the routing protocols used in the TCP/IP Internet.

Connections and Associations

We use the term *connection* to define the communication link between two processes. The term *association* is used for the 5-tuple that completely specifies the two processes that make up a connection:

{protocol, local-addr, local-process, foreign-addr, foreign-process}

The *local-addr* and *foreign-addr* specify the network ID and host ID of the local host and the foreign host, in whatever format is defined by the protocol suite. The *local-process* and *foreign-process* are used to identify the specific processes on each system that are involved in the connection, again in whatever format is defined by the protocol suite. In the TCP/IP suite, for example,

```
{tcp, 192.43.235.2, 1500, 192.43.235.6, 21}
```

could be a valid association, where the protocol is TCP, the local-address is 192.43.235.2, the local-process is 1500, the foreign-address is 192.43.235.6, and the foreign-process is 21. When we cover each protocol suite in the next chapter, we will see other examples of this 5-tuple.

We also define a *half association* as either *{protocol, local-addr, local-process}* or *{protocol, foreign-addr, foreign-process}*, which specify each half of a connection. This half association is also called a *socket* or a *transport address*. The term socket has been popularized by the Berkeley Unix networking system, where it is "an end point of communication," which corresponds to our definition of a half association. A socket pair then corresponds to our definition of an association. Unfortunately the XNS protocol suite uses the term socket to refer to what we called the 16-bit port above.

Client–Server Model

In Section 1.6 we briefly described the client–server model that is prevalent in computer networking. We defined *iterative servers* and *concurrent servers*. The former knows ahead of time about how long it takes to handle each request and the server process handles each request itself. In the latter case, the amount of work required to handle a request is unknown, so the server starts another process to handle each request.

Notice that the roles of the client and server processes are *asymmetric*. This means that both halves are coded differently. The server is started first and typically does the following steps:

1. Open a communication channel and inform the local host of its willingness to accept client requests on some well-known address.
2. Wait for a client request to arrive at the well-known address.
3. For an iterative server, process the request and send the reply. Iterative servers are typically used when a client request can be handled with a single response from the server.

   For a concurrent server, a new process is spawned to handle this client request. This requires a `fork` and possibly an `exec` under Unix, to start the new process. This new process then handles this client's request, and does not have to respond to other client requests. When this new process is finished, it closes its communication channel with the client, and terminates.
4. Go back to step 2: wait for another client request.

Implicit in the steps above is that the system somehow queues client requests that arrive while the server is processing a previous client's request. For example, with a concurrent server it is possible for additional requests to arrive during the time it takes for the server to spawn a new process to handle a request. Also implicit above is that the main server process exists as long as the host system is running—server processes never terminate unless forced to (such as when the system is being shut down).

The client process performs a different set of actions.

1. Open a communication channel and connect to a specific well-known address on a specific host (i.e., the server).
2. Send service request messages to the server, and receive the responses. Continue doing this as long as necessary.
3. Close the communication channel and terminate.

When the server opens a communication channel and waits for a client request, we call this a *passive open.* The client, on the other hand, executes what is called an *active open* since it expects the server to be waiting.

Who's Who in Networking Standards

There are many different organizations involved in the development of networking standards. Both the organizations and the standards are referred to by acronyms in most of the networking literature.

ISO, the *International Standards Organization*, was founded in 1946 and develops standards on a wide variety of subjects. The members of ISO are the national standards organizations from the member countries. The U.S. member of ISO is *ANSI*, the *American National Standards Institute.* The open systems interconnection model (OSI model) was adopted by ISO in 1984.

CCITT, the *Consultative Committee for International Telephony and Telegraphy*, develops standards for telephony and telegraphy. CCITT standards tend to be identified by a letter, followed by a period, followed by a number. For example, X.25 is a CCITT standard for packet-switched networks, and X.400 is a CCITT standard for electronic mail.

IEEE (pronounced "I triple-E"), the *Institute for Electrical and Electronic Engineers*, also issues standards. Its 802 standards are used for local area networks and provide standardization at the two lowest layers of the OSI model. The 802.2 standard defines the logical link control (LLC) used by the data-link layer. The 802.3, 802.4, and 802.5 standards define the media access control used by the data-link layer and the physical layer, for Ethernet†, token bus, and token ring networks, respectively.

EIA, the *Electronics Industry Association*, is a trade association concerned primarily with the physical layer of the OSI model. EIA developed the RS-232 standard that is used widely for connecting terminals to computers.

IAB, the *Internet Activities Board*, is a small group of researchers who direct most of the work on the TCP/IP protocols. It functions as a board of directors of the Internet.

Some networking standards originate with one organization and are then adopted by others. For example, the IEEE 802.2–1985 standard is also ISO Draft International Standard (DIS) 8802/2. The EIA RS-232 standard is also the CCITT V.24/V.28 standard.

Exercises

4.1 Draw a diagram of the local area network that you have access to. Include hosts, repeaters, bridges, routers, and gateways. Indicate what equipment (token ring, Ethernet, etc.) is being used at the physical layer.

† The original Ethernet standard was developed at Xerox Corp. in the 1970s and then standardized by DEC, Intel, and Xerox in 1981. The newest version is the IEEE 802.3 standard, dated 1985. It differs slightly from the original Ethernet.

5

Communication Protocols

5.1 Introduction

In this chapter we provide an overview of several communication protocol families. Our goal is to provide enough of an overview so that we understand how to use the protocols, and to provide references to more detailed descriptions of the actual design and implementation of the protocols.

We cover the following protocol suites, in the order shown:

- the TCP/IP protocol suite (the Internet protocols),
- Xerox Networking Systems (Xerox NS or XNS),
- IBM's Systems Network Architecture (SNA),
- IBM's NetBIOS,
- the OSI protocols,
- Unix-to-Unix Copy (UUCP).

For each protocol suite we give an overview, and then discuss the relevant portions for the protocol's layers. We start from the bottom layer and work our way up. Note that we cover each protocol family in its entirety, and we cover them in order of their importance in the remainder of this text.

5.2 TCP/IP—the Internet Protocols

5.2.1 Introduction

During the late 1960s and the 1970s the Advanced Research Projects Agency (ARPA) of the Department of Defense (DoD) sponsored the development of the ARPANET. The ARPANET included military, university, and research sites, and was used to support computer science and military research projects. (ARPA is now called DARPA, with the first letter of the acronym standing for "Defense.") In 1984 the DoD split the ARPANET into two networks—the ARPANET for experimental research, and the MILNET for military use. In the early 1980s a new family of protocols was specified as the standard for the ARPANET and associated DoD networks. Although the accurate name for this family of protocols is the "DARPA Internet protocol suite," it is commonly referred to as the TCP/IP protocol suite, or just TCP/IP.

In 1987 the National Science Foundation (NSF) funded a network that connects the six national supercomputer centers together. This network is called the *NSFNET*. Physically this network connects 13 sites using high-speed leased phone lines and this is called the NSFNET backbone. About eight more backbone nodes are currently planned. Additionally the NSF has funded about a dozen regional networks that span almost every state. These regional networks are connected to the NSFNET backbone, and the NSFNET backbone is also connected to the DARPA Internet. The NSFNET backbone and the regional networks all use the TCP/IP protocol suite.

There are several interesting points about TCP/IP.

- It is not vendor-specific.
- It has been implemented on everything from personal computers to the largest supercomputers.
- It is used for both LANs and WANs.
- It is used by many different government agencies and commercial sites, not just DARPA-funded research projects.

The DARPA-funded research has led to the interconnection of many different individual networks into what appears as a single large network—an *internet*, as described in the previous chapter. We refer to this internet as just the Internet (capitalized).

It is important to realize that while sites on the Internet use the TCP/IP protocols, many other organizations (with no government affiliation whatsoever) have established their own internets using the same TCP/IP protocols. At one extreme we have the Internet using TCP/IP to connect more than 150,000 computers throughout the United States, Europe and Asia, and at the other extreme we could have a network consisting of only two personal computers in the same room connected by an Ethernet using the same TCP/IP protocols.

To avoid having to qualify everything with "the TCP/IP protocol suite" or "the DARPA Internet protocol suite," we use the capitalized word *Internet*, as in "an Internet address," or "the Internet protocols," to refer to the TCP/IP protocol suite. This is to avoid confusion with internets that use protocols other than TCP/IP.

One reason for the increased use of the TCP/IP protocols during the 1980s was their inclusion in the BSD Unix system around 1982. This, along with the use of BSD Unix in technical workstations, allowed many organizations and university departments to establish their own LANs.

5.2.2 Overview

Although the protocol family is referred to as TCP/IP, there are more members of this family than TCP and IP. Figure 5.1 shows the relationship of the protocols in the protocol suite along with their approximate mapping into the OSI model.

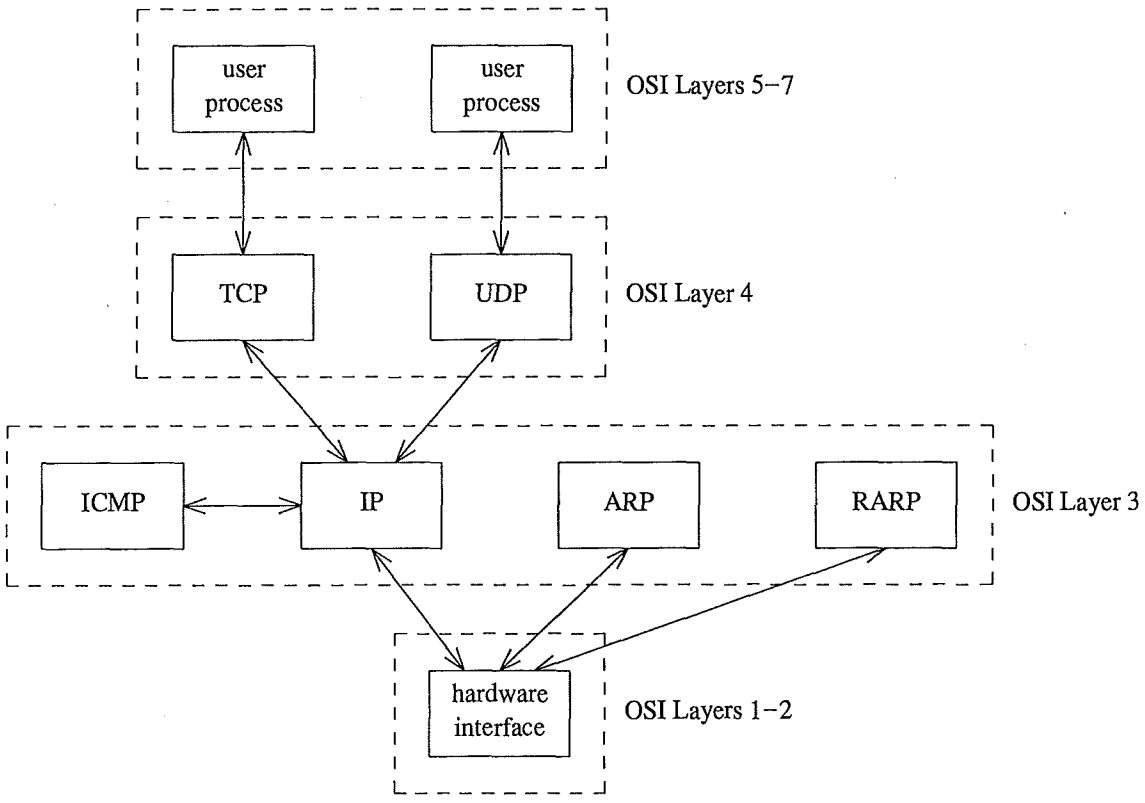


Figure 5.1 Layering in the Internet protocol suite.

TCP *Transmission Control Protocol.* A connection-oriented protocol that provides a reliable, full-duplex, byte stream for a user process. Most Internet application programs use TCP. Since TCP uses IP (as shown in Figure 5.1) the entire Internet protocol suite is often called the TCP/IP protocol family.

UDP *User Datagram Protocol.* A connectionless protocol for user processes. Unlike TCP, which is a reliable protocol, there is no guarantee that UDP datagrams ever reach their intended destination.

ICMP *Internet Control Message Protocol.* The protocol to handle error and control information between gateways and hosts. While ICMP messages are transmitted using IP datagrams, these messages are normally generated by and processed by the TCP/IP networking software itself, not user processes.

IP *Internet Protocol.* IP is the protocol that provides the packet delivery service for TCP, UDP, and ICMP. Note from the Figure 5.1 that user processes normally do not need to be involved with the IP layer.

ARP *Address Resolution Protocol.* The protocol that maps an Internet address into a hardware address. This protocol and the next, RARP, are not used on all networks. Only some networks need it.

RARP *Reverse Address Resolution Protocol.* The protocol that maps a hardware address into an Internet address.

There are other protocols in the Internet protocol suite that we do not consider in this text—GGP (Gateway-to-Gateway Protocol) and VMTP (Versatile Message Transaction Protocol), for example.

All the Internet protocols are defined by *Request for Comments (RFCs)*, which are their formal specifications. Corrections and explanations for many of the RFCs are found in RFC 1009 [Braden and Postel 1987], RFC 1122 [Braden 1989a], and RFC 1123 [Braden 1989b]. RFC 1118 [Krol 1989] provides hints for new members of the Internet community. It also describes how to obtain on-line information and how to be a good Internet neighbor.

Additional details on the TCP/IP protocol suite can be found in Comer [1988]. A sample implementation of UDP and IP is given in Comer [1987]. The source code for the complete 4.3BSD implementation of TCP/IP is provided on the BSD Networking Software release—refer to the Bibliography.

5.2.3 Data-Link Layer

The ARPANET consists of about 50 special purpose computers that are connected together using leased telephone lines at 57.6 Kbps. The host computers and gateways on the ARPANET are then connected to these special purpose computers. The 13 backbone sites of the NSFNET are connected with T1 leased phone lines, which operate at 1.544 Mbps. Most 4.3BSD systems using the TCP/IP suite for a LAN use Ethernet technology. Products also exist that use a token ring for a LAN using TCP/IP. There also exist implementations using an RS-232 serial line protocol (at speeds from 1200 bps to 19.2 Kbps) called the Serial Line Internet Protocol (SLIP). There are a variety of other data-link connections in use by TCP/IP networks: satellite links and packet radio, for example.

5.2.4 Network Layer—IP

IP Datagrams

The IP layer provides a connectionless and unreliable delivery system. It is connectionless because it considers each IP datagram independent of all others. Any association between datagrams must be provided by the upper layers. Every IP datagram contains the source address and the destination address (described below) so that each datagram can be delivered and routed independently. The IP layer is unreliable because it does not guarantee that IP datagrams ever get delivered or that they are delivered correctly. Reliability must also be provided by the upper layers. The IP layer computes and verifies a checksum that covers its own 20-byte header (that contains, for example, the source and destination addresses). This allows it to verify the fields that it needs to examine and process. But if an IP header is found in error, it is discarded, with the assumption that a higher layer protocol will retransmit the packet.

As we saw in the gateway examples from the previous chapter, it is the IP layer that handles routing through an Internet. The IP layer is also responsible for fragmentation, as described in the previous chapter. For example, if a gateway receives an IP datagram that is too large to transmit across the next network, the IP module breaks up the datagram into fragments and sends each fragment as an IP packet. (Technically, the protocol unit exchanged by the end-to-end IP layers is an IP datagram. An IP datagram can be fragmented into smaller IP packets. When fragmentation does occur, the IP layer duplicates the source address and destination address into each IP packet, so the resulting IP packets can be delivered independently of each other.) The fragments are reassembled into an IP datagram only when they reach their final destination. If any of the fragments are lost or discarded, the entire datagram is discarded by the destination host.

The IP layer provides an elementary form of flow control. When IP packets arrive at a host or gateway so fast that they are discarded, the IP module sends an ICMP source quench message to the original source informing that system that the data is arriving too fast. With 4.3BSD, for example, the ICMP source quench is passed to the TCP module (assuming it was a TCP message that caused the source quench), which then decreases the amount of data being sent on that connection.

The reference for IP is RFC 791 [Postel 1981a].

Internet Addresses

Every protocol suite defines some type of addressing that identifies networks and computers. An Internet address occupies 32 bits and encodes both a network ID and a host ID. The host ID is relative to the network ID. Every host on a TCP/IP internet must have a unique 32-bit address. TCP/IP addresses on the Internet are assigned by a central authority—the Network Information Center (NIC) located at SRI International. Even if you are establishing your own internet, and don't plan to join the Internet, you can still get an assigned Internet address by contacting SRI. (See Appendix M of Nemeth, Snyder, and Seebas [1989] for the details on how to do this.)

A 32-bit Internet address has one of the four formats shown in Figure 5.2.

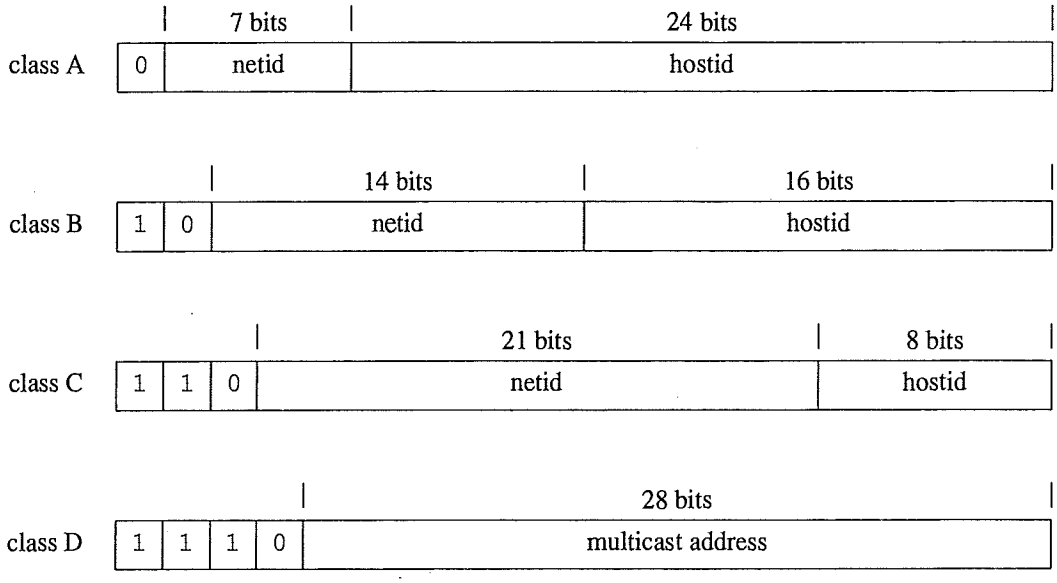


Figure 5.2 Internet address formats.

Class A addresses are used for those networks that have a lot of hosts on a single network, while Class C addresses allow for more networks but fewer hosts per network. For network addresses assigned by the NIC, only the type of address (Class A, B, or C) and the network ID is assigned. The requesting organization then has responsibility for assigning individual host addresses on that network. We won't consider multicast addresses (Class D) any further.

Internet addresses are usually written as four decimal numbers, separated by decimal points. Each decimal digit encodes one byte of the 32-bit Internet address. For example, the 32-bit hexadecimal value 0x0102FF04 is written as 1.2.255.4. This example is a Class A address with a network ID of 1 and a host ID of 0x02FF04. A sample Class B address is 128.3.0.5, and a sample class C address is 192.43.235.6.

Every IP datagram contains the 32-bit Internet address of the source host and the 32-bit Internet address of the destination host, in every 20-byte IP header. Since an Internet address is comprised of a network ID and a host ID, gateways can easily extract the network ID field from a 32-bit address and route IP datagrams based solely on the network ID. This is an important concept for routing, as it means that a gateway needs only to know the location of other networks, and does not to need to know the location of every host on an Internet.

Recall that a multihomed host is connected to two or more networks. This implies that it must have two or more Internet addresses, one for each network that it is connected to. This means that every Internet address specifies a unique host, but each host does not have a unique address.

Subnet Addresses

Any organization with an Internet address of any class can subdivide the available host address space in any way it desires, to provide subnetworks. For example, if you have a Class B address, there are 16 bits allocated for the host ID. If your organization wants to assign host IDs to its 150 hosts, that are in turn organized into 10 physical networks, there are two different ways to do this.

- You can allocate host IDs of 1 through 150, ignoring the physical network structure. This requires that all the gateway systems among the 150 hosts know where each individual host is located, for routing purposes. Adding a new host requires that each gateway's routing table be updated.
- You can allocate some of the high-order bits from the host ID, say the high-order 8 bits, for the network ID within your subnetwork. These 8 bits are independent of the Class B network ID. The remaining 8 bits of the Class B host ID you then use to identify the individual hosts on each internal network. Using this technique, your gateway systems can extract the 8-bit internal network ID and use it for routing, instead of having to know where each of the 150 hosts are located. Adding a new host on an existing internal network doesn't require any changes to the internal gateways.

A picture of what the second option is doing is given in Figure 5.3.

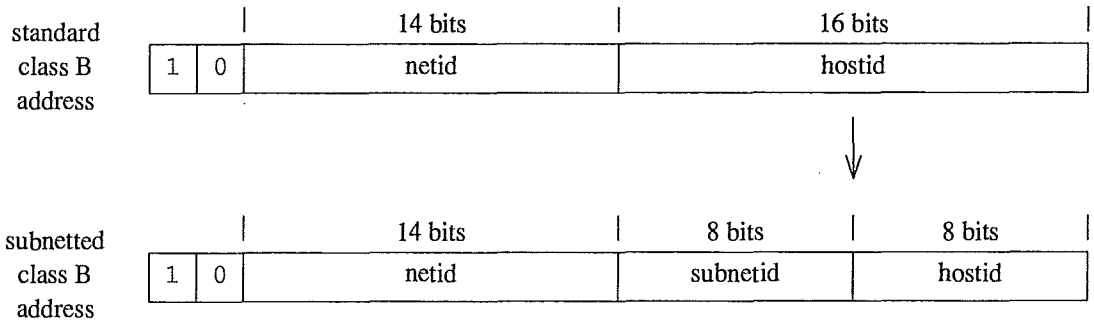


Figure 5.3 Class B Internet address with subnetting.

This feature adds another level to the Internet address hierarchy.

- network ID
- subnet ID within network
- host ID within subnet

The reference for subnetting is RFC 950 [Mogul and Postel 1985].

Address Resolution

If we have an Ethernet LAN consisting of hosts using the TCP/IP protocols, we have two types of addresses: 32-bit Internet addresses and 48-bit Ethernet addresses. Recall from the previous chapter that the 48-bit Ethernet addresses are typically assigned by the manufacturer of the interface board and are all unique. We have the following address resolution problems:

- If we know the Internet address of the other host that we want to communicate with, how does the IP layer determine which Ethernet address corresponds to that host? This is the *address resolution problem.*
- When a diskless workstation is initialized (bootstrapped), the operating system can usually determine its own 48-bit Ethernet address from its interface hardware. But we do not want to embed the workstation's 32-bit Internet address into the operating system image, as this prevents us from using the same image for multiple workstations. How can the diskless workstation determine its Internet address at bootstrap time? This is the *reverse address resolution problem.*

The first problem is solved using the Internet *Address Resolution Protocol (ARP)*, and the second uses the Internet *Reverse Address Resolution Protocol (RARP).*

The ARP allows a host to broadcast a special packet on the Ethernet that asks the host with a specified Internet address to respond with its Ethernet address. Every host on the Ethernet receives this broadcast packet, but only the specified host should respond. Once the requesting host receives the response it can maintain the mapping between the Internet address and the Ethernet address for all future packets destined for the same Internet address. The reference for ARP is RFC 826 [Plummer 1982].

The RARP is intended for a LAN with diskless workstations. One or more systems on the LAN are the RARP servers and contain the 32-bit Internet address and its corresponding 48-bit Ethernet address for each workstation. This allows the operating system for each workstation to be generated without having to have its Internet address as part of its configuration. When the workstation is initialized, it obtains its 48-bit Ethernet address from the interface hardware and broadcasts an Ethernet RARP packet containing its Ethernet address and asking for its Internet address. Every host on the Ethernet LAN receives this broadcast, but only the RARP servers should respond. The reference for RARP is RFC 903 [Finlayson et al. 1984a].

5.2.5 Transport Layer—UDP and TCP

User processes interact with the TCP/IP protocol suite by sending and receiving either TCP data or UDP data. The relationship of TCP and UDP to our simplified 4-layer model from Section 1.5 is shown in Figure 5.4.

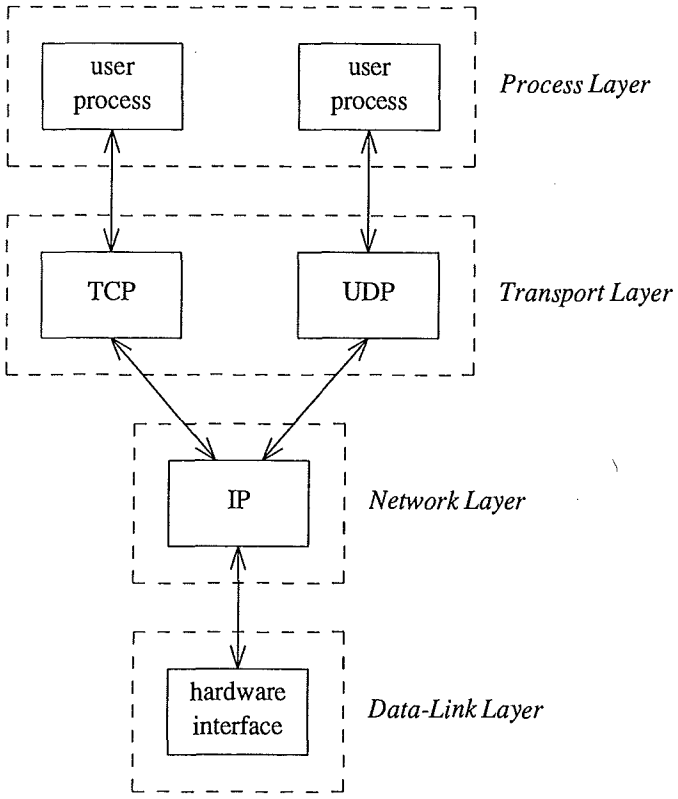


Figure 5.4 4-layer model showing UDP, TCP, and IP.

These two protocols are sometimes referred to as TCP/IP or UDP/IP, to indicate that both use IP also.

TCP provides a connection-oriented, reliable, full-duplex, byte-stream service to an application program. UDP, on the other hand, provides a connectionless, unreliable datagram service. Figure 5.5 compares IP, UDP, and TCP against the modes of service that we described in the previous chapter.

	IP	UDP	TCP
connection-oriented ?	no	no	yes
message boundaries ?	yes	yes	no
data checksum ?	no	opt.	yes
positive ack. ?	no	no	yes
timeout and rexmit ?	no	no	yes
duplicate detection ?	no	no	yes
sequencing ?	no	no	yes
flow control ?	no	no	yes

Figure 5.5 Comparison of protocol features for IP, UDP, and TCP.

Notice that we list IP as not having flow control, since the source quench feature described earlier is not an end-to-end flow control technique.

Since the IP layer provides an unreliable, connectionless delivery service for TCP, it is the TCP module that contains the logic necessary to provide a reliable, virtual circuit for a user process. TCP handles the establishment and termination of connections between processes, the sequencing of data that might be received out of order, the end-to-end reliability (checksums, positive acknowledgments, timeouts), and the end-to-end flow control. RFC 793 [Postel 1981c] is the standard reference for TCP.

UDP provides only two features that are not provided by IP: port numbers (described below) and an optional checksum to verify the contents of the UDP datagram. But these two features are enough reason for a user process to use UDP instead of trying to use IP directly, when a connectionless datagram protocol is required. The reference for UDP is RFC 768 [Postel 1980].

Port Numbers

Figure 4.14 shows that it is possible for more than one user process at a time to be using either TCP or UDP. This requires some method for identifying the data associated with each user process. Both TCP and UDP use 16-bit integer *port numbers* for this identification.

When a client process wants to contact a server, the client must have a way of identifying the server that it wants. If the client knows the 32-bit Internet address of the host on which the server resides it can contact that host, but how does the client identify the particular server process? To solve this problem, both TCP and UDP have defined a group of *well-known ports*. (These are the Internet-specific *well-known addresses* that we described in the previous chapter.) For example, every TCP/IP implementation that supports FTP, the File Transfer Protocol, assigns the well-known port of 21 (decimal) to it. TFTP, the Trivial File Transfer Protocol, is assigned the UDP port of 69.

Let's take this client–server interaction to the next step and assume that a client sends a message to the FTP server on some host by sending a message to port 21 on that host. How does the FTP server know where to send its response? First, the server can obtain the 32-bit Internet address of the client from the IP datagram, since these datagrams contain the source and destination Internet addresses in the 20-byte IP header. The client process also requests an unused port number from the TCP module on its local host. The server can obtain the 16-bit port number from the TCP header. As long as the client's TCP module does not reassign this port number to some other process before the first client is finished, there won't be any conflict. When TCP or UDP assign unique port numbers for user processes, they are called *ephemeral port numbers* (short lived). The process that receives the ephemeral port number (the client process in this example) doesn't care what value it is. It is the other end of the communication link (the server)

that needs it. TCP and UDP port numbers in the range 1 through 255 are reserved. All the well-known ports are in this range. Some operating systems reserve additional ports for privileged programs (4.3BSD reserves the ports 1–1023 for superuser processes as we describe in Section 6.8), and the ephemeral ports are above these reserved ports.

We have described a hierarchical addressing scheme that involves multiple layers.

- The IP datagram contains the source and destination Internet addresses in its IP header. These two 32-bit values uniquely identify the two host systems that are communicating.
- Also contained in the IP header is a protocol identifier, so that the IP module can determine if an IP datagram is for TCP, UDP, or some other protocol module that uses IP, which isn't discussed in this text.
- The UDP header and the TCP header both contain the source port number and the destination port number. These two 16-bit integer values are used by the protocol modules to identify a particular user process. Note that the TCP ports are independent of the UDP ports, since the IP header specifies the protocol. TCP port 1035, for example, is independent of UDP port 1035.

As described in the previous chapter, the addition of this control information by the different protocol modules is *encapsulation*. The combination of information from different sources using identifiers such as port numbers, protocol types, and Internet addresses is called *multiplexing*.

The 5-tuple that defines an *association* in the Internet suite consists of

- the protocol (TCP or UDP),
- the local host's Internet address (a 32-bit value),
- the local port number (a 16-bit value),
- the foreign host's Internet address (a 32-bit value),
- the foreign port number (a 16-bit value).

An example could be

```
{tcp, 128.10.0.3, 1500, 128.10.0.7, 21}
```

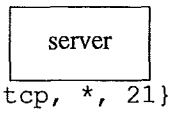
Figure 5.6 diagrams the encapsulation that takes place with UDP data on an Ethernet.

If the length of the IP datagram (the data, plus the UDP header, plus the IP header) is greater than the MTU of the network access layer, then the IP layer has to fragment the datagram before it is passed to the network access layer. If this happens, the receiving IP layer has to reassemble the fragments into a single datagram before it is passed to the upper layers (e.g., UDP). Whether fragmentation takes place or not, the size of the data message (the datagram) exchanged by the two UDP layers is the same.

As with UDP, it is also possible for the TCP layer to pass messages to the IP layer that exceed the underlying network's MTU. If this happens, the sending IP layer does fragmentation and the receiving IP layer does reassembly. For performance reasons, however, most TCP implementations try to prevent IP fragmentation.

Concurrent Servers

With a concurrent server, what happens if the child process that is spawned by the main server continues to use its well-known port number while servicing a long request? Let's examine a typical sequence. First, the server is started on the host `orange` and it does a passive open using its well-known port number (21, for this example). It is now waiting for a client request.



We use the notation `{tcp, *, 21}` to indicate that the server is waiting for a TCP connection on any connected network (i.e., interface), on port 21. If the host on which the server is running is connected to more than one TCP/IP network, the server can specify that it only wants to accept a connection from a client on one specific network. Our asterisk notation indicates that a connection will be accepted on any network.

At some later time a client starts up and executes an active open to the server. The client has an ephemeral port number assigned to it by the protocol module (TCP, in this example). Assume the ephemeral port number is 1500 for this example, and assume that the name of the client's host is `apple`. This is shown in Figure 5.8.

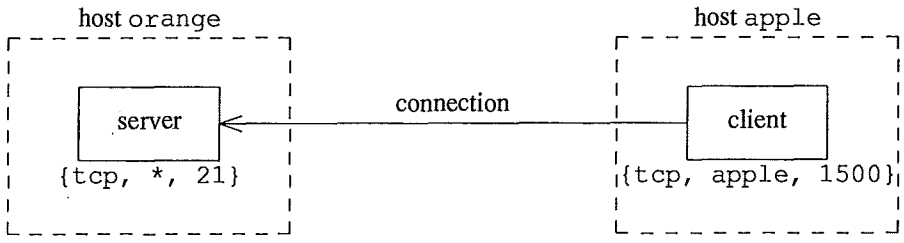


Figure 5.8 Connection from client to server.

The notation `{tcp, apple, 1500}` is the client's *half association* or *socket*. Here we are designating the Internet address as `apple`, using the name of the host instead of its 4-digit notation. When the server receives the client's connection request, it `forks` a copy of itself, passing the connection to the child process, as shown in Figure 5.9.

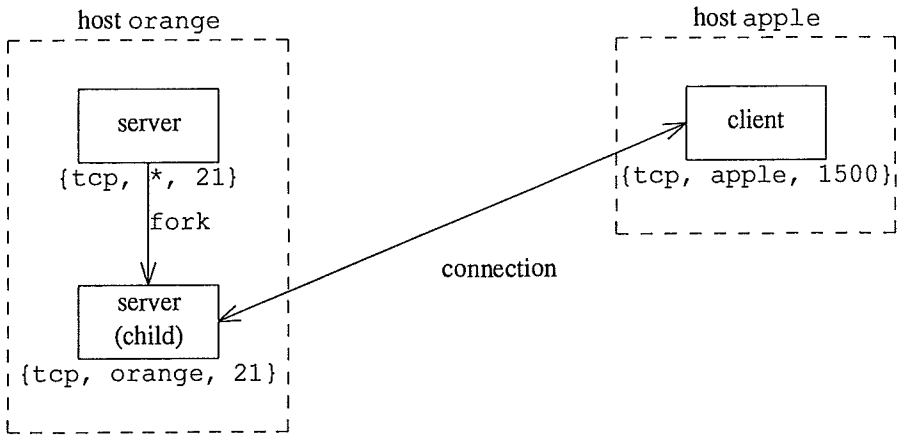


Figure 5.9 Concurrent server passes connection to child.

The association is `{tcp, orange, 21, apple, 1500}`. The server process that was waiting for the client connection now returns to its wait loop, letting the child process handle the client's request.

Assume that another client process on the host `apple` requests a connection to the same server. The client's TCP module assigns it a new ephemeral port number, say 1501, so that the half association `{tcp, apple, 1501}` is unique on the host `apple`. This gives us the picture shown in Figure 5.10.

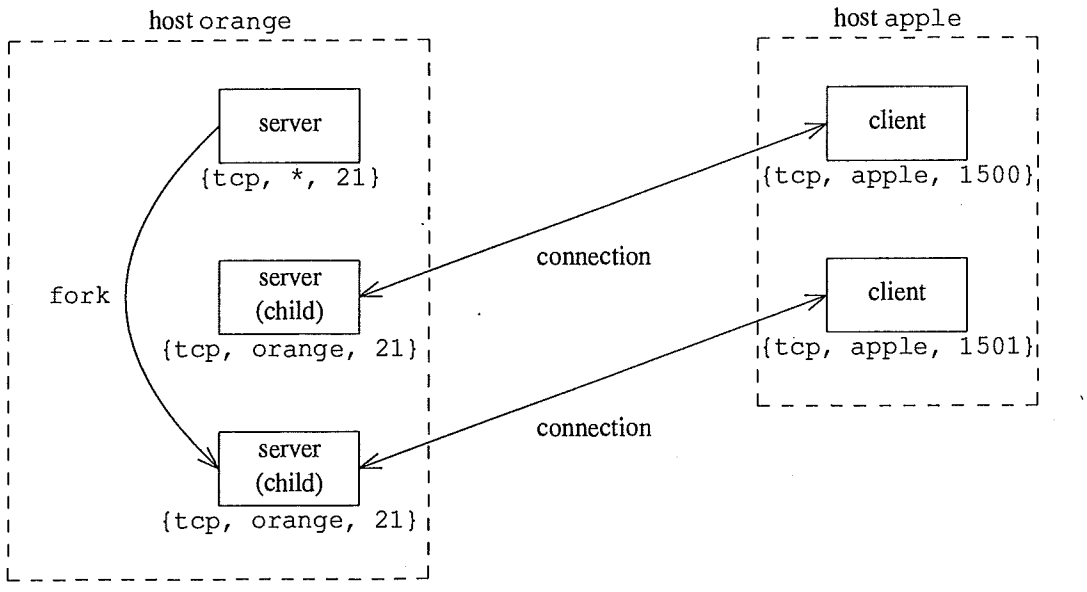


Figure 5.10 Second client connection passed to another child.

Note that even though there are two identical half associations on the `orange` host, `{tcp, orange, 21}`, the two complete associations are unique.

```
{tcp, orange, 21, apple, 1500}
{tcp, orange, 21, apple, 1501}
```

The TCP module on the `orange` system is able to determine which server child process is to receive a given data message, based on the source Internet address and the source TCP port number.

We assumed in the examples above that the TCP module assigns a unique ephemeral port number to a process (e.g., a client) if the process does not need to assign a well-known port to itself (e.g., a server). This is what normally happens. There are instances, however, when a process can request that a specific port be used. The Internet FTP has this requirement, and it is interesting to examine this as another example of Internet addressing and port assignment.

FTP normally runs using the standard client and concurrent-server relationship, as shown above. User commands are passed from the client to the server across the TCP connection, with the server's responses being returned on the connection. But when we request a file to be transferred between the client and server, another TCP connection is established between the two processes. This provides one connection for commands and one connection for data. The client initiates the transfer by sending a command across the existing connection to the server. The client also creates a new communication channel, using the same port number that it is using for the control connection. The client does a passive open on this new channel, waiting for the server to complete the connection. This is different from the previous examples, as the client now has to tell its TCP module that it is OK to use the same port number, even though the half association, `{tcp, apple, 1500}`, for example, is not unique on the client's host. (This is done using the `SO_REUSEADDR` socket option, as we'll describe in Section 6.11.) To guarantee a unique association when the server completes the connection with this new client port, the server must use some other port. FTP dictates that the server use the well-known port 20 for this purpose. If we assume that both the client processes in our previous examples are FTP clients, and both have established a data connection with their respective servers, we have the picture shown in Figure 5.11. All four associations are unique, as required by the Internet protocol suite.

```
{tcp, orange, 20, apple, 1500}
{tcp, orange, 21, apple, 1500}
{tcp, orange, 20, apple, 1501}
{tcp, orange, 21, apple, 1501}
```

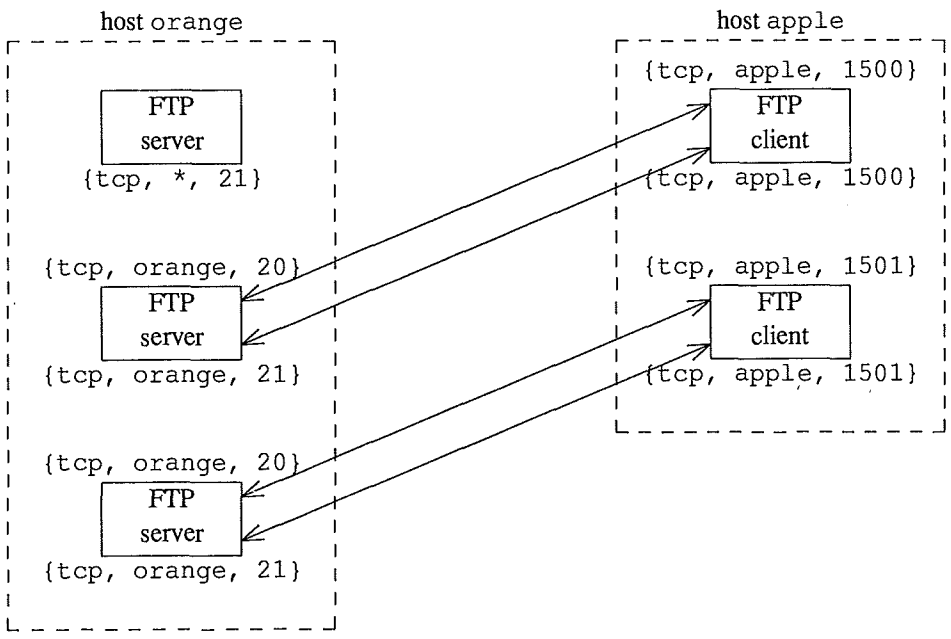


Figure 5.11 FTP example with two clients and two servers.

Buffering and Out-of-Band Data

UDP is a datagram service. Every datagram written by a user process has a header encapsulated by the UDP layer, then the datagram is passed to the IP layer for transmission. There is no reason for the UDP layer to let the user's data hang around in the UDP layer. The data is transmitted as soon as the IP layer can get to it, so the concepts of buffering and out-of-band data do not apply to UDP.

TCP, however, presents a byte-stream service to the user process, and provides both of these features. In TCP terminology the flush output command is called the *push* flag, and out-of-band data is called *urgent data*. By definition, TCP supports any amount of out-of-band data, but not all implementations support more than one byte at a time of urgent data.

We show how a user process can set the TCP push flag with a socket option in Section 6.11, and we return to TCP out-of-band data in Section 6.14.

Buffer Sizes and Limitations

By definition the maximum size of an IP datagram is 65,536 bytes. Assuming a 20-byte IP header, this leaves up to 65,516 bytes for other data in the datagram. Realize, however, that as soon as the size of an IP datagram exceeds the size of the underlying network's MTU, fragmentation occurs. Furthermore, not all TCP/IP implementations support IP fragmentation, let alone fragmentation of a 65,536-byte datagram. Every TCP/IP implementation must support a minimum IP datagram size of 576 bytes. Note that a network can have an MTU smaller than 576, but any host must be able to

reassemble the fragmented IP packets into at least a 576-byte IP datagram. The actual maximum size of an IP datagram depends on the IP software on both ends, as well as the software on every gateway between the two ends.

Since UDP packets are transmitted using IP, if the result of adding the UDP header and the IP header causes the datagram to exceed the network's MTU, fragmentation occurs. This means, for example, that sending 2048-byte UDP packets on an Ethernet guarantees fragmentation.

TCP is different, since it breaks up the data into what it calls *segments*. The segment size used by TCP is agreed on between the two ends when a connection is established. In Section 6.11 we describe a socket option that allows a user process to determine the segment size being used by TCP.

Cabrera et al. [1988] show the effects of IP fragmentation on network performance. Kent and Mogul [1987] provide arguments for not allowing fragmentation at all.

5.2.6 Application Layer

There are several application programs that are provided by almost every TCP/IP implementation. One strength of the TCP/IP protocol suite is the availability of these standard applications for a variety of operating environments.

Additional details on FTP, TELNET, and SMTP are given in Stallings et al. [1988], in addition to the RFCs referenced below.

FTP—File Transfer Protocol

FTP is a program used to transfer files from one system to another. It provides a rich set of features and options, such as user authentication, data conversion, directory listings, and the like. Its standard definition is RFC 959 [Postel and Reynolds 1985].

A typical sequence of events is for an interactive user to invoke an FTP client process on the local system. This client process establishes a connection with an FTP server process on the remote system using TCP. FTP establishes two connections between the client and server processes—one for control information (commands and responses) and the other for the data being transferred. The interactive user is prompted for access information on the remote system (login name and password, if required), and files can then be transferred in both directions. FTP handles both binary and text files.

TFTP—Trivial File Transfer Protocol

TFTP is a simpler protocol than FTP. While it provides for file transfer between a client process and a server process, it does not provide user authentication or some of the other features of FTP (listing directories, moving between directories, etc.). TFTP uses UDP, not TCP. The reference for TFTP is RFC 783 [Sollins 1981].

We develop a complete client and server implementation of TFTP in Chapter 12.

TELNET—Remote Login

TELNET provides a remote login facility. It allows an interactive user on a client system to start a login session on a remote system. Once a login session is established, the client process passes the user's keystrokes to the server process. As with the FTP program, TELNET uses TCP. The standard for TELNET is RFC 854 [Postel and Reynolds 1983].

We consider the design of a remote login facility in Chapter 15 when we discuss the 4.3BSD remote login command.

SMTP—Simple Mail Transfer Protocol

SMTP provides a protocol for two systems to exchange electronic mail using a TCP connection between the two systems. The protocol definition for SMTP is RFC 821 [Postel 1982]. The standard for the format of the mail messages is RFC 822 [Crocker 1982]. RFC 974 [Partridge 1986] specifies how mail is routed.

5.3 XNS—Xerox Network Systems

5.3.1 Introduction

Xerox Network Systems (also called Xerox NS, or XNS) is the network architecture developed by Xerox Corporation in the late 1970s for integrating their office products and computer systems. XNS is an *open* system, in that Xerox has published and made available the protocols used by XNS. Other vendors also provide hardware and software that supports the XNS protocols. Most 4.3BSD systems provide support for the XNS protocol suite. XNS is similar in structure to the TCP/IP protocol suite, so the presentation in this section assumes the reader has read the TCP/IP description in the previous section.

5.3.2 Overview

The arrangement of the layers in the XNS protocol suite, and their approximate mapping into the OSI model is shown in Figure 5.12.

ECHO — *Echo Protocol.* A simple protocol that causes a host to echo the packet that it receives. Most XNS implementations support this protocol.

RIP — *Routing Information Protocol.* A protocol used to maintain a routing database for use on a host in the forwarding of IDP packets to another host. Typically a routing process exists on the host, and this process uses RIP to maintain the database.

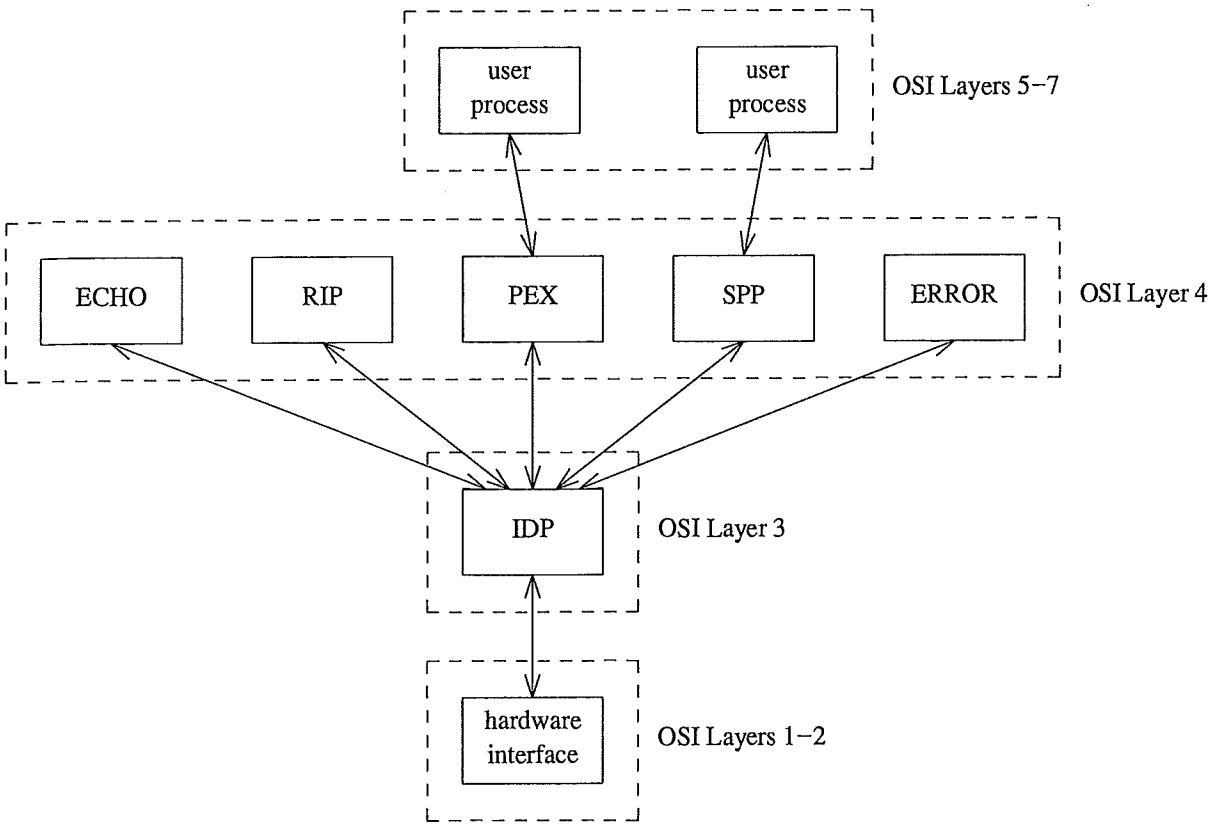


Figure 5.12 Layering in the XNS protocol suite.

PEX — *Packet Exchange Protocol.* An unreliable, connectionless, datagram protocol for user processes. Although PEX is not a reliable protocol, it does retransmission but does not perform duplicate detection. We describe PEX later in this section.

SPP — *Sequenced Packet Protocol.* A connection-oriented, reliable protocol for user processes. It provides a byte stream for the user process with optional message boundaries. SPP is the most commonly used protocol in the XNS suite, similar to TCP in the Internet suite. We describe SPP in more detail later in this section.

ERROR — *Error Protocol.* A protocol that can be used by any process to report that it has discovered an error and therefore discarded a packet.

IDP — *Internet Datagram Protocol.* IDP is the connectionless, unreliable datagram protocol that provides the packet delivery service for all the above protocols. We describe IDP later in this section.

The five protocols that we show at OSI layer 4 are termed the "Internet" Transpo Protocols by Xerox. These are defined by Xerox [1981b], with additional details give in O'Toole, Torek, and Weiser [1985]. Boggs et al. [1980] gives some insight into th design of XNS.

Most Xerox applications are built using the Courier remote procedure call protoco Courier, in turn, is built using SPP. We show this in Figure 5.13.

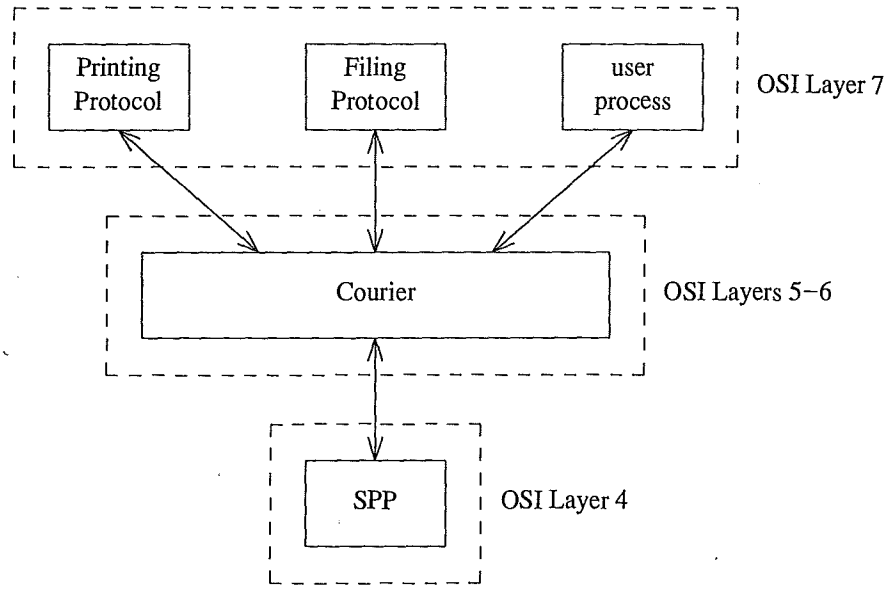


Figure 5.13 Typical XNS applications that use Courier.

A user process has the choice of interacting directly with SPP or using Courier. We return to Courier in Chapter 18 when we describe remote procedure calls in more detail.

Note that all XNS protocols use IDP. Recall from the TCP/IP protocol suite that ARP and RARP did not use IP.

5.3.3 Data-Link Layer

The typical XNS network is an Ethernet, although other technologies such as leased or switched telephone lines using SDLC are possible.

5.3.4 Network Layer—IDP

XNS Addresses

An XNS address occupies 12 bytes and is comprised of three parts:

- a 32-bit network ID,
- a 48-bit host ID,
- a 16-bit port number.

(Unfortunately, the XNS literature calls the 16-bit port number a *socket*. To avoid confusion with the term socket that we defined in the previous chapter, we call this XNS field a *port*.) In our definition of a socket from the previous chapter, *{protocol, local-addr, local-process}*, the *protocol* is `xns`, the *local-addr* is the combination of the network ID and the host ID, and the *local-process* is the port number. Note that an association in the TCP/IP protocol suite interprets the *local-process* relative to the *protocol*, which is why we have to differentiate between TCP and UDP. With XNS, however, that distinction isn't required since the *local-process* (the port number) does not depend on any specific transport protocol within XNS.

The host ID is an absolute number that must be unique across all XNS internets. Typically the 48-bit host ID is set to the 48-bit Ethernet address, as most Xerox internets are built using Ethernets. (Recall from the previous chapter that all Ethernet interfaces are provided by their manufacturer with a unique 48-bit Ethernet address.) With unique host IDs, the network ID is redundant, but it is required for routing purposes. Like the host ID, the network ID must also be unique across all XNS internets. Both host IDs and network IDs are assigned by Xerox Corp. to guarantee their uniqueness. Host IDs are typically written as six hexadecimal digits separated by periods, as in 2.7.1.0.2a.19. The network ID is typically written as a decimal integer.

Recall that TCP/IP host IDs are relative to the network ID, not absolute. For example, the host ID in both of the Class B addresses 128.1.0.3 and 128.2.0.3 is 3, but the first network ID is 1 and the second is 2. XNS host addresses, on the other hand, form a flat address space.

IDP Packets

Everything in XNS is eventually transmitted in an IDP packet. IDP provides a connectionless and unreliable delivery service, similar to the IP layer in the TCP/IP protocol suite. Every IDP packet contains a 30-byte header with the following fields:

- source XNS address (host ID, network ID, port),
- destination XNS address (host ID, network ID, port),
- checksum,
- length of data (typically 0–546 bytes),
- higher layer packet type (SPP, PEX, etc.).

The differences between IP and IDP are as follows:

- The IDP packet contains a checksum that includes the entire IDP packet. Recall that the IP checksum is only for the IP header, and does not include the data in the IP datagram. IP forces the upper layers (UDP and TCP) to add their own checksums, if desired.
- IDP packets contain a 16-bit port number. In the TCP/IP suite, both UDP and TCP are required to define their own port numbers, which are stored in their own headers. The IDP port number is in the IDP header.

- IDP demultiplexes the incoming datagrams based on the port number, since the IDP port number is independent of the actual protocol. The IP layer, on the other hand, demultiplexes the incoming datagrams based on their protocol field—usually TCP or UDP. The TCP and UDP layers then demultiplex the data based on the port numbers in the TCP and UDP headers.

Given the first two points, it is more reasonable for a user process to use IDP directly, when an unreliable datagram protocol is required, than for a user process to use IP directly.

Since the IDP layer demultiplexes the incoming datagrams based on the local port number, and not the packet type, it is possible for a protocol module to receive packets of another protocol. For example, it is possible for the SPP protocol module to receive an ERROR protocol packet whose destination is the port using SPP. Each protocol package in the XNS suite must be prepared to receive ERROR protocol packets. Again, this differs from the TCP/IP suite.

5.3.5 Transport Layer—SPP and PEX

SPP—Sequenced Packet Protocol

SPP, the predominant transport layer protocol in XNS, is similar to TCP. It provides a connection-oriented, reliable, full-duplex service to an application program. Unlike TCP, which provides only a byte-stream interface, there is a three-level hierarchy in the data being transferred using SPP.

- Bytes are the basic entity.
- A packet is composed of zero or more bytes.
- A message is composed of one or more packets.

Using these data forms, SPP can present three different interfaces to a user process.

- A *byte-stream* interface: the bytes are delivered to the user process in order. The user process reads or writes the data using any convenient buffer size. Message boundaries are preserved, but there are no packet boundaries seen by the user process.
- A *packet-stream* interface: the packets are delivered to the user process in order. The user process reads or writes entire packets, each of which is passed on to the IDP layer. The user process has to know more details, such as the format of the

12-byte SPP header, to set and detect message boundaries. The user process is not to associate any semantics with the physical packet boundaries that it reads and writes; instead the message boundaries available through SPP are to be used for message demarcation.

- A *reliable-packet* interface: the packets are delivered to the user process, but they might be out of order. This interface presents the packets to the user process as they arrive, which might be out of order, since two consecutive IDP packets might travel from the source to the destination following different routes. Duplicate packets are removed by the SPP software.

The first two interfaces are typically provided, while the reliable-packet interface is an option. Also, it is possible for both ends of a connection to be using a different interface. For example, there is nothing to prevent a client from using the byte-stream interface when communicating with a server that is using the packet-stream interface. We compare these three interfaces in Figure 5.14.

	SPP		
	byte-stream	packet-stream	reliable-packet
connection-oriented ?	yes	yes	yes
message boundaries ?	yes	yes	no
packet boundaries ?	no	yes	yes
data checksum ?	yes	yes	yes
positive ack. ?	yes	yes	yes
timeout and rexmit ?	yes	yes	yes
duplicate detection ?	yes	yes	yes
sequencing ?	yes	yes	no
flow control ?	yes	yes	yes

Figure 5.14 Comparison of interfaces provided by SPP.

In this figure we have added the line "packet boundaries" to differentiate between the byte-stream interface and the packet-stream interface. In future figures we show only the SPP byte-stream interface, since it supports message boundaries. Since the reliable-packet interface is not common, we do not consider it any further in this text.

In Figure 5.15 we diagram the encapsulation that takes place with SPP data on an Ethernet.

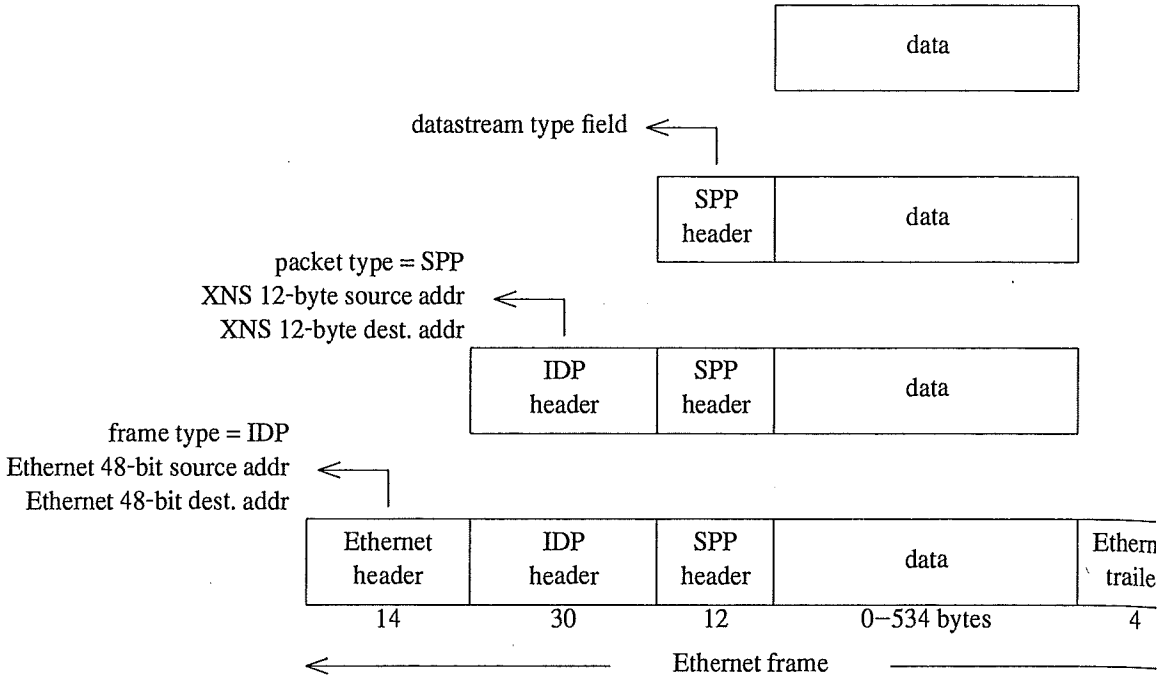


Figure 5.15 Encapsulation of SPP data on an Ethernet.

The datastream type field and the packet type field are termed *bridge fields* and w describe them later in this section. The 12-byte XNS address fields contain the 48-bi host address, the 32-bit network address and the 16-bit port number.

PEX—Packet Exchange Protocol

Another transport layer protocol is PEX the Packet Exchange Protocol. PEX is a data gram oriented protocol, similar to UDP, but PEX retransmits a request when necessary (requiring a timeout and retransmission) and PEX does not do duplicate detection. PEX has a 6-byte header that includes a 32-bit ID field and a 16-bit client type. The protoco operates as follows:

1. The client sets the ID field to some value that it chooses and sets the client type field to a value that specifies the type of service requested. The packet is sent to the server.
2. The server performs the service specified by the client type.
3. When the server forms its response packet, it must return the same ID field that was sent by the client. This way, when the client receives the response it knows which request the server is responding to. The client must then use a different ID field for its next request.

The purpose of the client type field in the PEX header is to allow a single server to handle multiple service requests, each service corresponding to a different client type. The PEX module on the client end will retransmit the request if a response if not received in a specified amount of time, obviating the user's client process from having to do this.

The encapsulation that takes place with PEX data on an Ethernet is shown in Figure 5.16.

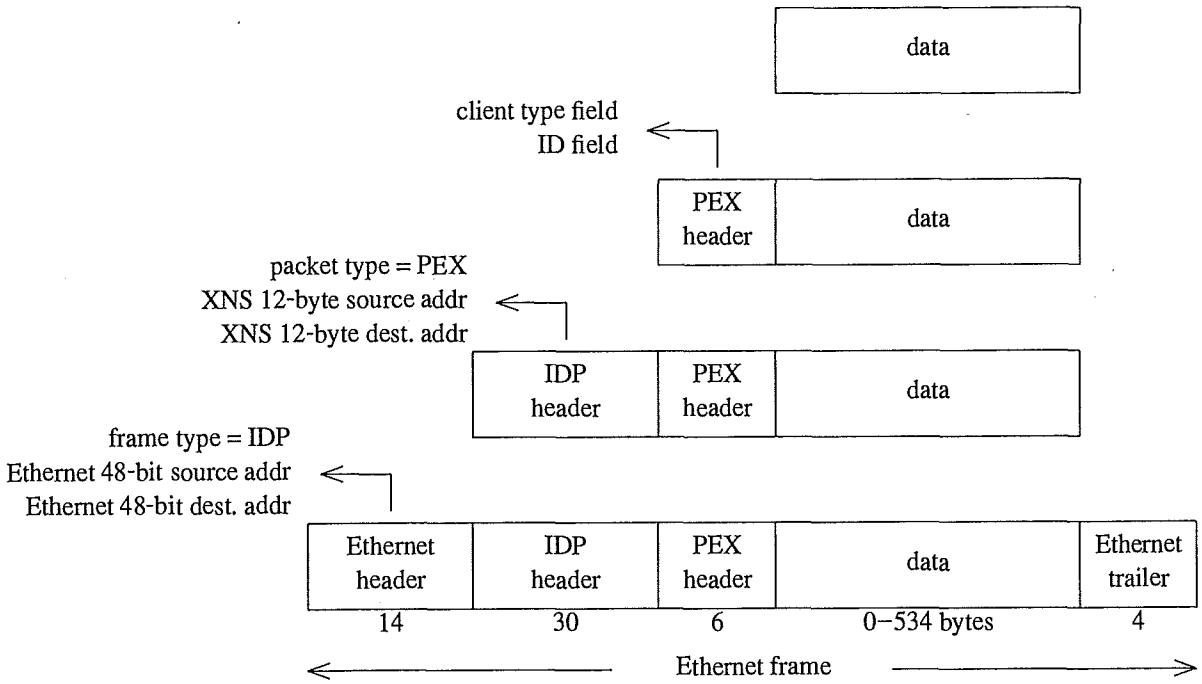


Figure 5.16 Encapsulation of PEX data on an Ethernet.

The relationship of the XNS protocols that we have described to our simplified 4-layer model is shown in Figure 5.17.

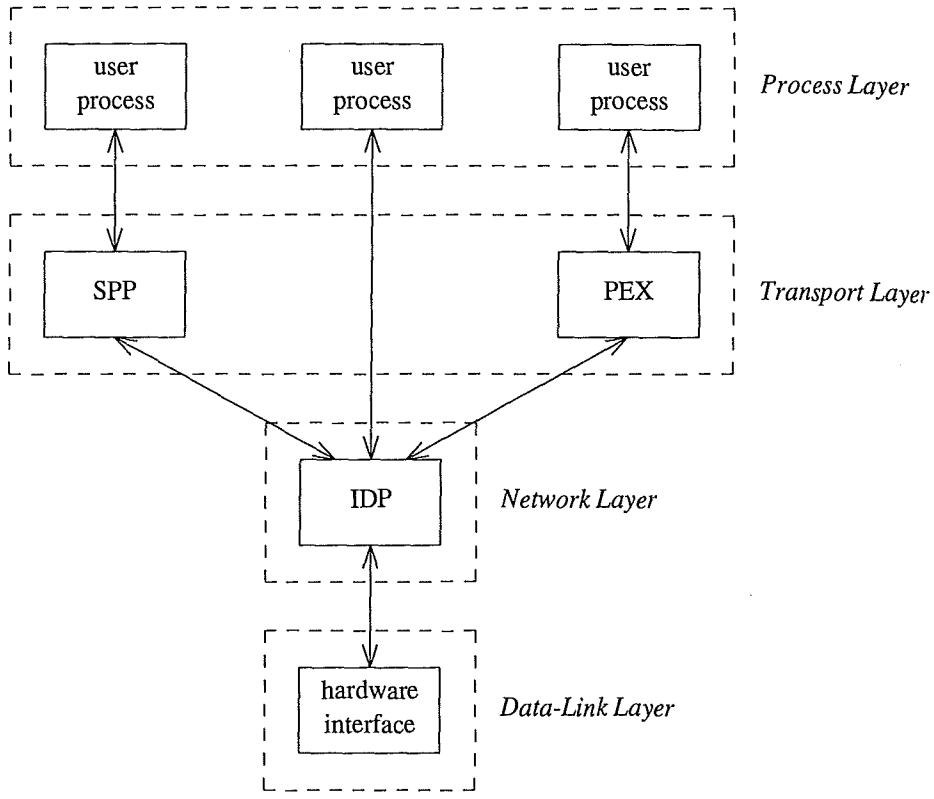


Figure 5.17 4-layer model showing SPP, PEX, and IDP.

In Figure 5.18 we add IDP, PEX, and SPP to our table of protocol services from the previous section.

	IP	IDP	UDP	PEX	TCP	SPP
connection-oriented ?	no	no	no	no	yes	yes
message boundaries ?	yes	yes	yes	yes	no	yes
data checksum ?	no	yes	opt.	yes	yes	yes
positive ack. ?	no	no	no	no	yes	yes
timeout and rexmit ?	no	no	no	yes	yes	yes
duplicate detection ?	no	no	no	no	yes	yes
sequencing ?	no	no	no	no	yes	yes
flow control ?	no	no	no	no	yes	yes

Figure 5.18 Comparison of protocol features: Internet protocols and XNS protocols.

As mentioned earlier in this section, the column for SPP is for either the byte-stream interface or the packet-stream interface, since both provide the same features for the services listed.

XNS Bridge Fields

The XNS protocols define *bridge fields* in the headers that are encapsulated by each layer. These fields are not used by a particular layer, but are set and interpreted by the next higher layer. There are three bridge fields.

1. The *packet type* field in the IDP header is not used by the IDP layer. It specifies the protocol of the data in the IDP packet. This is how, for example, the SPP module knows that a received packet is an ERROR protocol packet and not an SPP data packet.
2. The *datastream type* field in the SPP header is not used by the SPP module, but is available for the user process that is using SPP. The XNS reference, Xerox [1981b], provides an example use of this field in shutting down an SPP connection between two user processes.
3. The *client type* field in the PEX header is not used by the PEX protocol module. Again, it is intended for the user process that is using PEX.

The complication imposed by these bridge fields is that they are in the headers that are encapsulated and decapsulated by the protocol software—they are not in the normal data area that the user reads and writes. This requires some form of interface between the protocol modules (SPP and PEX) and the user process, to allow us to set and examine these bridge fields.

We'll examine the 4.3BSD socket options that allow a user process to examine and set these bridge fields in Section 6.11. Additionally, the XNS echo client that we develop in Section 11.3 uses one of these socket options to set the packet type field in the IDP header.

Buffering and Out-of-Band Data

PEX is a datagram service, so the concepts of buffering and out-of-band data don't apply. SPP, however, provides a single byte of out-of-band data. The Xerox specification states that this byte of data is to be made available to the user process as quickly as possible, and furthermore it is presented to the user process again according to its normal sequence. This means that the receiving SPP layer makes a copy of the out-of-band byte in some special location and also puts the byte into its sequential position in the receiver's input buffer.

We'll return to SPP out-of-band data in Section 6.14.

Buffer Sizes and Limitations

IDP packets are not fragmented as Internet IP datagrams might be. Most Xerox systems enforce a maximum IDP packet size of 576 bytes. Allocating 30 bytes for the IDP header, this allows up to 546 bytes of data in an IDP packet. Allowing 12 bytes for an SPP header, the maximum SPP packet size is 534 bytes.

5.3.6 Application Layer

The main use of XNS today is to communicate with Xerox hardware. As mentioned earlier, most of these applications use the Courier remote procedure facility, which we describe in Chapter 18. Xerox standards are available for the predominant application protocols—printing protocol (sending files to a Xerox printer), and filing protocol (sending and receiving files from a Xerox file server), for example. Xerox [1985] provides additional information on some of these applications.

5.4 SNA—Systems Network Architecture

5.4.1 Introduction

SNA, Systems Network Architecture, was originally released by IBM in 1974. It has since been enhanced many times and is now the predominant method used to form internets of various IBM computers. Additionally, many vendors other than IBM provide various levels of SNA support, allowing many non-IBM systems to participate in SNA networks. SNA is an architecture, not a product. There exist software and hardware products from both IBM and other vendors that implement different portions of SNA.

Our reason for covering SNA in a book on Unix networking, is that most vendors of Unix systems now support SNA, in one form or another. Frequently the support is for LU 6.2, which is the emphasis of this section. Unfortunately there is no common programming interface across all these products, so we won't be able to say much about the C interface to an SNA implementation.

SNA was originally designed to support the networking of nonprogrammable devices, such as terminals (termed "workstations" by IBM) and printers, to IBM mainframes. (An IBM "mainframe" is a system based on the System/370 architecture, as opposed to other IBM systems, both large and small.) Because of this non peer-to-peer evolution, where one system (the mainframe) is always the master in the communication relationship, SNA appears more complicated than the other communication architectures that we've discussed (TCP/IP, for example). SNA has always had a terminology all its own, and has been confounded by the mainframe requirements just to implement and configure an SNA network—CICS, VTAM, NCP, cluster controllers, and the like. Only in the past five years has SNA adopted the protocols and implementations that allow two user processes to communicate easily with each other, and without the requirement of a mainframe in the communication link.

The user of a network is defined to be an *end user*, which is either a user at a terminal or an application program (process). An end user interacts with a *logical unit* (referred to as an *LU*) to access the network. The type of services provided by the LU to the end user differs for each type of LU. LU types 2, 3, 4, and 7 support communication between processes and terminals, while LU types 1, 6.1, and 6.2 are for communication between two processes. The LU in turn interacts with a *physical unit* (referred to as a *PU*). The PU is, in essence, the operating system of the node, and it controls the data-link connections between the node and the network. We picture this in Figure 5.19.

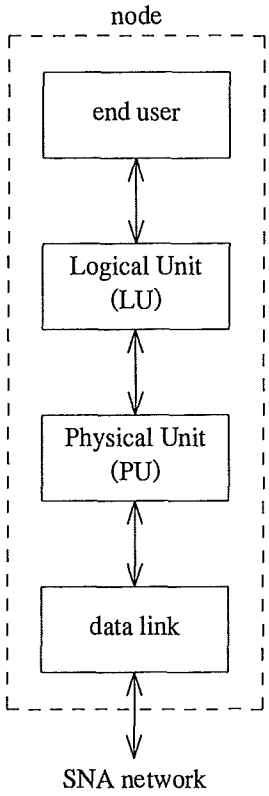


Figure 5.19 SNA node with logical unit and physical unit.

If we consider the PU as the operating system of the node, then the data-link portion is the device driver of the node. The type of node is specified by the PU type. Five types of nodes are currently supported—1, 2.0, 2.1, 4, and 5. These are referred to, for example, as a "Type 2.1 node," or "T2.1 node," or as PU 2.1. What we have called earlier in this text a "network host" or a "system on a network," is referred to by IBM as a *node*—an addressable unit on a network. IBM uses the term node, since an SNA node can be a nonprogrammable device such as a terminal or printer.

Our interest in this text is only with LU 6.2 and PU 2.1, as these are the strategic IBM products, and the only SNA implementations that adequately support communication between processes.

The bibliography lists several SNA manuals available from IBM that provide additional details on all the features mentioned here.

5.4.2 Overview

SNA was developed in a layered approach that is similar to the OSI model. The seven SNA layers are shown in Figure 5.20.

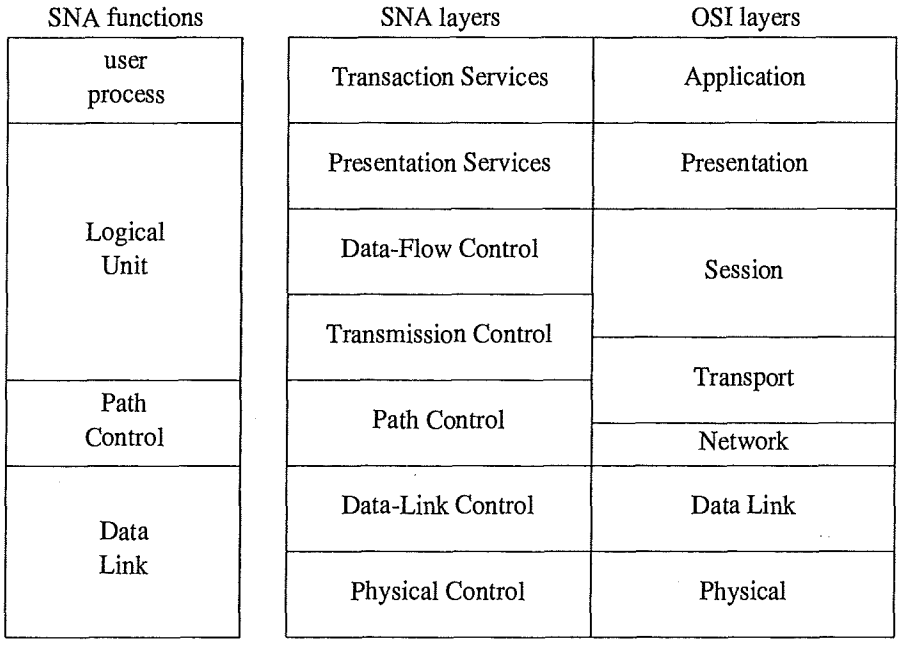


Figure 5.20 7-layer SNA model and approximate mapping between SNA layers and OSI model.

Since SNA predates the OSI model, there is no exact mapping from the seven SNA layers into the seven OSI layers. An approximate mapping between the two, along with a consolidation of the seven SNA layers into the corresponding SNA functions, is also shown in Figure 5.20.

In the seven SNA layers, the LU performs the functions of layers 4, 5, and 6—transmission control, data-flow control, and presentation services. The top layer, transaction services, is what we have called the user process. The bottom two layers are combined into what we have called the network access layer, as these are the two layers that change depending on the interface types supported by the node (SDLC, token ring, etc.).

Note that the four SNA functions in Figure 5.20 correspond to our simplified 4-layer model. What we call a user process is called a transaction program (TP) by IBM.

Two forms of SNA networks have evolved.

- *Subarea SNA networks.*
  These networks are built around IBM mainframes that maintain centralized control over the network. Internets of independent SNA subarea networks can be formed using the SNA network interconnect (SNI) facility.
- *APPN networks.*
  APPN stands for "Advanced Peer-to-Peer Networking" and was introduced by IBM in 1986 for the System/36. It has also been called SNA/LEN (low entry networking). In 1988 the capability was introduced for APPN networks to be connected together through an SNA subarea network. APPN is limited to PU Type 2.1 nodes and supports LU 6.2. APPN has features not found in subarea networks, such as dynamic routing. Baratz et al. [1985] describe the prototype implementation of APPN.

Today most of the more than 20,000 SNA networks in existence are subarea networks based on IBM mainframes. But the increased shift towards smaller systems and the increased capabilities of APPN should lead to the creation of many APPN-based networks.

5.4.3 Data-Link Layer

The predominant form of network access used in SNA networks is SDLC—synchronous data-link control. Typical SDLC links operate between 4.8 Kbps and 57.6 Kbps over leased or switched telephone lines. The token ring is now starting to be used for LANs. One IBM product, the RT, supports LU 6.2 over an Ethernet. IBM mainframes can communicate with each other using SNA over a System/370 data channel.

5.4.4 Path-Control Layer

In the TCP/IP and XNS protocol suites, the IP layer and the IDP layer both provided an unreliable datagram delivery service to the layers above. In SNA we encounter a different scenario where the SNA path-control layer provides a virtual circuit service to its upper layer (the LU). This means that the path-control layer, and the data-link layer beneath it, provide error control, flow control, and sequencing.

Path control is the layer responsible for moving packets around in an SNA network. Figure 5.21 shows the encapsulation that each layer applies to a message of user data being transmitted by an SNA node, assuming an SDLC data link.

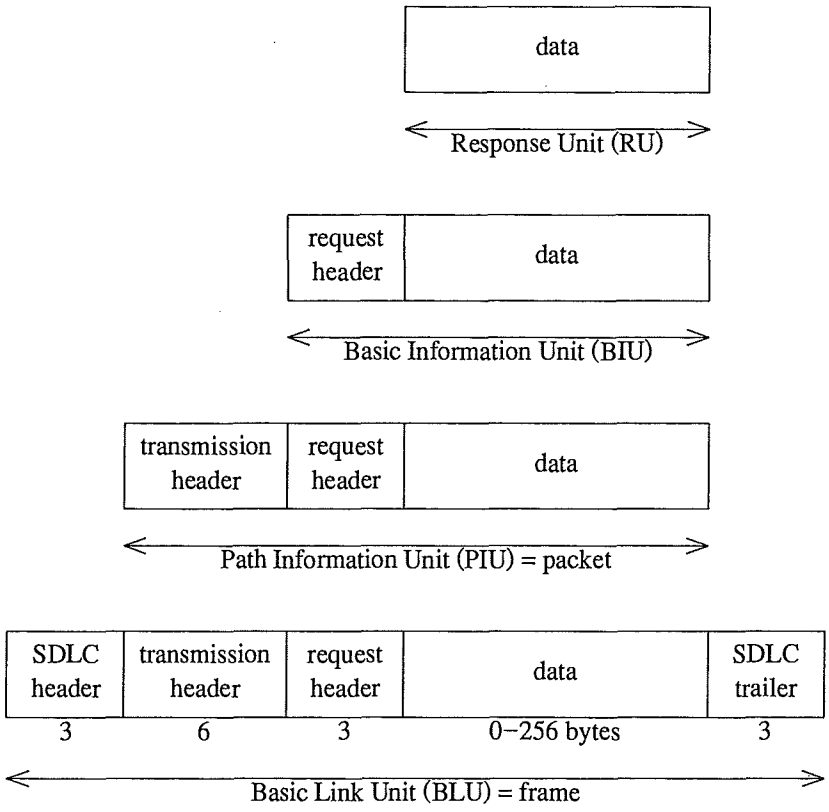


Figure 5.21 Encapsulation of SNA data on an SDLC link.

The request header is the LU header, and the transmission header is the PC (path control) header, if we follow the conventions from the TCP/IP and XNS protocol suites where the name of an encapsulated header is the name of the layer that adds the header. There is another header that is sometimes added by the LU—a function management header (FMH) that carries control information between LUs. This header is placed between the request header and the data, with a bit turned on in the request header specifying that a function management header is present.

In Figure 5.21 we show the data portion as being from 0 to 256 bytes in length. The maximum response unit size is typically 256 or 512 bytes, but this size can be negotiated to other values by the two LUs.

Every LU in a given SNA network must have a unique name, from 1 to 8 characters. In an internet of SNA networks, every network must have a unique name, also from 1 to 8 characters. A particular LU in an internet is identified by its *network-qualified LU name*. This consists of

- network name,
- LU name within the network.

The network-qualified LU name is written as *netname.LUname* using a period between the two names. The actual mapping of this network-qualified LU name to a physical address is handled by a portion of the LU called *directory services*.

The size of the transmission header that path control uses for moving packets through a network ranges from 6 to 26 bytes, depending on the node types. We showed a 6-byte transmission header Figure 5.21. These 6-byte headers are used for traffic between two adjacent Type 2.1 nodes. The 26-byte transmission header is used for data exchanges between two adjacent subarea nodes that support explicit routing and virtual routing. A complete treatment of SNA routing and SNA address formats is beyond the scope of this text. Fortunately, it is not a requisite to understanding the use of LU 6.2, which is the intent of our coverage of SNA.

Path control also does packet fragmentation and reassembly, termed *segmenting* in SNA, when a packet must be divided into pieces before being passed to the data-link layer. This is similar to the fragmentation and reassembly done by the IP layer, in TCP/IP.

5.4.5 LU 6.2—APPC

LU 6.2 is also referred to as *APPC* for *Advanced Program-to-Program Communication* and was released by IBM in 1982. LU 6.2 can use either PU 2.0 or PU 2.1. The major difference is that PU 2.0 can only be used to communicate with an IBM mainframe, and has inherent in it the master–slave relationship, where the mainframe is the master. PU 2.1, however, is newer, and systems implementing PU 2.1 can communicate with each other, without the need for a mainframe in the communication path. All newer IBM implementations of SNA (System/36, System/38, AS/400, RT, PC) support PU 2.1.

LU 6.2 provides a connection-oriented, reliable, half-duplex service to an application program. Note that this is the first half-duplex service that we have encountered. Both TCP and SPP are full-duplex protocols. LU 6.2, however, allows data flow between the user processes only in a single direction at a time. In Figure 5.22 we add LU 6.2 to the table of protocol features that we have been building. This time, we'll add a row for the connection-oriented protocols indicating whether the protocol provides a full-duplex data stream to the user process.

	IP	IDP	UDP	PEX	TCP	SPP	LU6.2
connection-oriented ?	no	no	no	no	yes	yes	yes
message boundaries ?	yes	yes	yes	yes	no	yes	yes
data checksum ?	no	yes	opt.	yes	yes	yes	no
positive ack. ?	no	no	no	no	yes	yes	yes
timeout and rexmit ?	no	no	no	yes	yes	yes	yes
duplicate detection ?	no	no	no	no	yes	yes	yes
sequencing ?	no	no	no	no	yes	yes	yes
flow control ?	no	no	no	no	yes	yes	yes
full-duplex ?					yes	yes	no

Figure 5.22 Comparison of protocol features: Internet, XNS, and SNA.

Notice that LU 6.2 does not provide an end-to-end data checksum, which is different

from the other connection-oriented protocols we've covered—TCP and SPP. The developers of SNA consider the hop-by-hop reliability of the data-link layer (the cyclic redundancy check provided by the SDLC protocol, for example) to be an adequate test for corrupted data. (We return to this topic in the last section of this chapter.) Yet LU 6.2 provides positive acknowledgments, timeout and retransmission, duplicate detection, and sequencing, so we still consider it a "reliable" protocol.

Sessions and Conversations

In SNA terminology, the peer-to-peer connection between two user processes is called a *conversation.* The peer-to-peer connection between two LUs is called a *session.* A session is usually a long-term connection between two LUs, while a conversation is often of a shorter duration. This is shown in Figure 5.23.

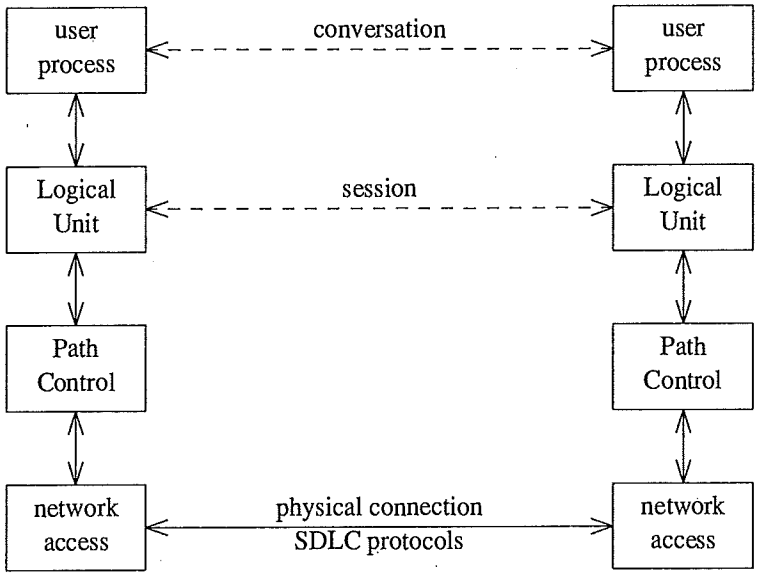


Figure 5.23 SNA sessions and conversations.

Using Figure 5.23 we can list some of the features provided by a Type 2.1 node that are not available with a Type 2.0 node.

- A T2.1 node supports multiple hardware links.
- A T2.1 node supports *parallel sessions* for its LUs. This means that two LUs, each in a different node, can have multiple sessions with each other.
- A T2.1 node supports *multiple sessions* for its LUs. This means that a given LU can have sessions with more than one partner LU at the same time.

An example is shown in Figure 5.24.

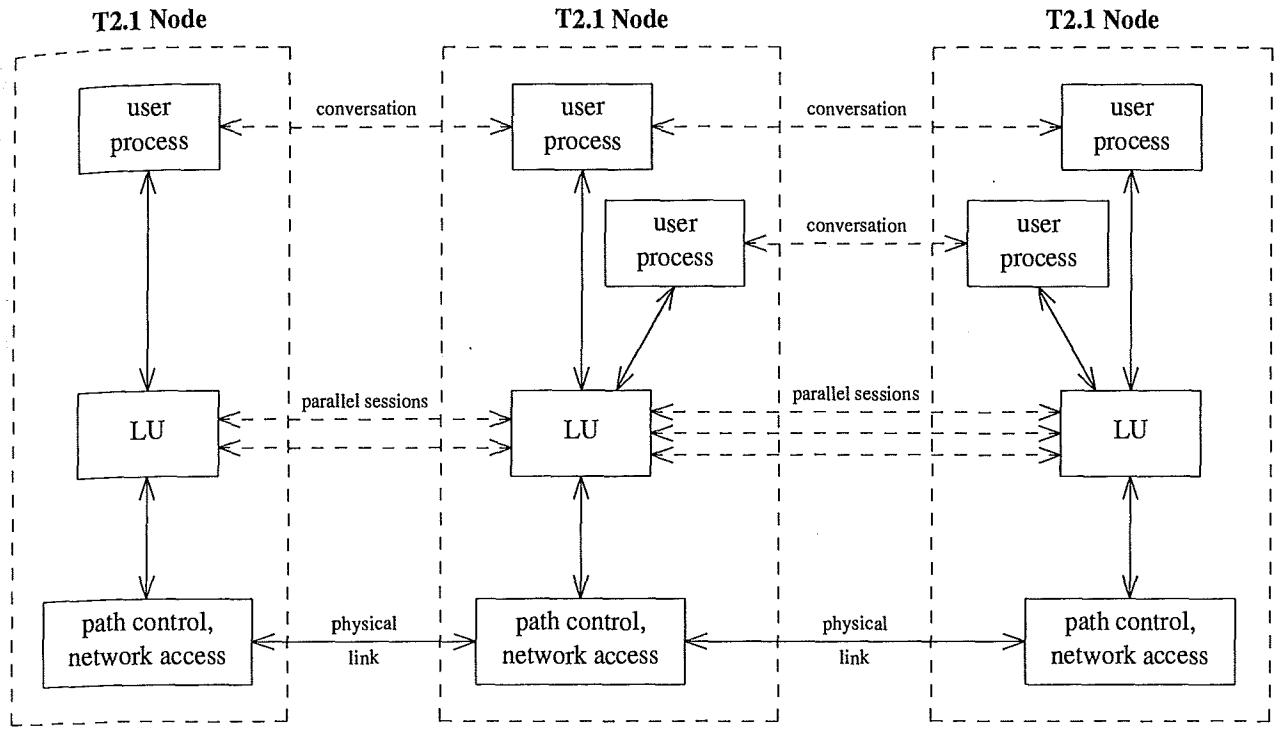


Figure 5.24 Example of sessions and conversations between Type 2.1 nodes.

Sessions are expensive to establish, so a typical LU establishes a certain number of sessions with its partner LUs. This forms a pool of active sessions for the LU to manage. When a user process wants to allocate a conversation, it requests the use of a session from the LU for the conversation. The LU picks an available session from its pool and dedicates this session to the conversation. The user process can use this conversation for a single transaction with its partner process, or it can hold on to the conversation for a long time. When the process is finished with the conversation, the process deallocates it. This allows the LU to put the conversation back into an available pool.

Using the terminology from the previous chapter we would call a conversation a *connection*, and ignore the fact that a conversation takes place over a session. Indeed, sessions are usually established by a network operator or when the system is initialized, so the typical user has no control over the session allocations. Note that LU 6.2 differs from both TCP and SPP in its explicit allocation and use of sessions. While a TCP connection between two user processes implies that the two TCP modules have a "session" of some form between them, we do not have to preallocate a certain number of sessions between two TCP modules. There is no inherent limit on the number of "sessions" that a TCP module can have with other TCP modules, other than any resource limits in its implementation (table sizes, number of processes allowed by the operating system, etc.). LU 6.2 requires more user control in specifying the number of sessions to allocate between LUs.

In our client–server model, we identified a server process as a process that had executed a passive open, awaiting a connection from a client process. This implied that the

server process was already executing when it did the passive open. The server was identified by a well-known address that was protocol-dependent. LU 6.2 is different in that the client specifies the *name* of the partner program to execute—called the TP (transaction program) name. Furthermore, the target LU usually creates a new program instance (i.e., a new process) for every conversation that is started with a given TP name.

Each LU associates a conversation with a unique identifier, which we call a conversation ID, and this ID associates a packet with a particular process. (Since a conversation has exclusive use of a session, we can call this either a conversation ID or a session ID.) In an APPN network the conversation ID is a 16-bit integer. This 16-bit value is carried in the transmission header and is used by APPN to specify the user process. From our definition of an *association* from the previous chapter, for SNA it consists of

- the protocol (LU 6.2),
- the local network-qualified LU name,
- the local conversation ID,
- the foreign network-qualified LU name,
- the foreign conversation ID.

An example of this 5-tuple could be

```
{lu62, NET1.LUJOE, 5, NET1.LUMIKE, 3}
```

Again, we are not concerned with the conversion of a network-qualified LU name, such as `NET1.LUJOE`, into its internal representation as an SNA address.

LU 6.2 Protocol Boundary

The interface between a user process and the LU is called "presentation services" in SNA. There are two interfaces possible for a user process to LU 6.2:

- mapped conversations,
- unmapped conversations.

An unmapped conversation is also called a basic conversation. The major difference between the two types of conversations is in the format of the data that is exchanged between the process and the LU (which we discuss later in this section).

The interface between a user process and LU 6.2 is defined as a collection of *verbs* that a transaction program can execute to request a service from the LU. This defines the protocol boundary between the program and LU 6.2. The actual mapping of these verbs into an application program interface (API) depends on the specific LU 6.2 software being used. An API could be a set of functions that a user process can call. As mentioned earlier, there is no standard API for LU 6.2, not even among different IBM products. Therefore, instead of describing an actual API, we'll describe the different LU 6.2 verbs that most LU 6.2 implementations provide to an application program.

Here we provide a brief summary of the LU 6.2 verbs. We list only the names of the mapped conversation verbs—unless otherwise noted, a basic verb that does a similar function is available, and its name is formed by removing the MC_ prefix. For example, ALLOCATE is the basic conversation equivalent of MC_ALLOCATE.

MC_ALLOCATE	Allocates a conversation with another program. Required arguments are the name of the program to execute and the name of the LU where that program is located. The process issuing the ALLOCATE starts in the send state and the partner process starts in the receive state.
MC_CONFIRM	Sends a confirmation request to the remote process and waits for a reply. This allows the two processes in a conversation to synchronize their processing with each other.
MC_CONFIRMED	Sends a confirmation reply to the remote process.
MC_DEALLOCATE	Deallocates a conversation.
MC_FLUSH	Forces the transmission of the local send buffer to the other LU. In general, LU 6.2 waits until a full buffer is available before sending anything to the other process.
MC_GET_ATTRIBUTES	Obtains information about a conversation, such as the name of the partner LU.
MC_PREPARE_TO_RECEIVE	Changes the conversation from send to receive state. Recall that LU 6.2 is a half-duplex protocol, not full-duplex.
MC_RECEIVE_AND_WAIT	Waits for information to be received from the partner process. The information returned can be user data or a confirmation request.
MC_RECEIVE_IMMEDIATE	Receives any information that is available in the local LU's buffer, but does not wait for information to arrive.
MC_REQUEST_TO_SEND	Notifies the partner process that this process wants to send data. When the local process receives a "send" indication from the partner process, the state of the conversation changes.

MC_SEND_DATA	Sends one data record to the partner process. For a mapped conversation, one user data record is optionally mapped into a mapped conversation record (MCR), as described below. For an unmapped conversation, some portion of a logical record is sent. Note that regardless of the conversation type, this verb might not cause any data to be sent to the other process, depending on the buffering in the local LU. See the MC_FLUSH verb above.
MC_SEND_ERROR	Informs the partner process that the local process has detected an application error of some form.

There are additional verbs defined by LU 6.2 that are considered "control operator verbs." These are used to start sessions, change the number of sessions, stop sessions, and the like. Their usage is product dependent and we don't consider them in this text.

The LU 6.2 protocol boundary defines numerous return codes from each of the verbs. For example, the SEND_DATA verb can return that all is OK and additionally it can indicate that the partner process has issued a REQUEST_TO_SEND.

Logical Records and GDS Variables

All user process data consists of a 2-byte length field (LL) followed by zero or more bytes of data. This is called a *logical record* and is shown in Figure 5.25.

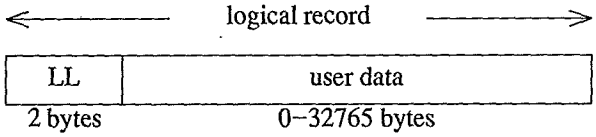


Figure 5.25 SNA logical record.

The length includes the two bytes occupied by the length field, so its value is always greater than or equal to two.†

Two processes using the basic conversation verbs exchange logical records. A process that is writing logical records need not write a complete logical record with every SEND_DATA verb. That is, a logical record consisting of 500 bytes (an LL field of 500, followed by 498 bytes of user data) can be written in three pieces—100 bytes, followed by a SEND_DATA of 250 bytes, followed by a SEND_DATA of 150 bytes. Similarly, the receiving process can read the logical record (RECEIVE_AND_WAIT) in whatever sized chunks it desires. A process must write a complete logical record, based on the LL value.

† Some internal LU components exchange control information for a session with the partner LU using length fields less than two. This "invalid" length field informs the LU that the record is a control record and not user data. We don't consider these control records in this text.

In the above example with an LL of 500, the process cannot write 400 bytes and then do a read. Not writing a complete logical record is called *truncation* and generates an error indication for the receiving process.

For mapped conversations each `MC_SEND_DATA` verb executed by the process generates a single *user data record.* If a mapping is being done, the mapping is applied to the user data record, generating the *mapped conversation record (MCR).* A 2-byte LL field and a 2-byte ID field are prepended to the beginning of the MCR, generating a logical record. This is shown in Figure 5.26.

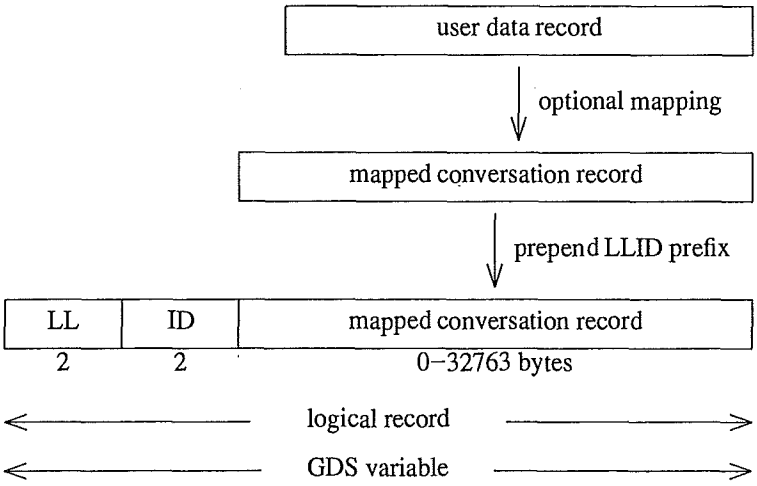


Figure 5.26 SNA mapped conversation record.

This logical record is the same as the logical record that is written using the basic conversation verbs, except a 2-byte ID field follows the LL field. This guarantees that the length of a logical record from an `MC_SEND_DATA` verb is greater than or equal to 4 bytes (since the lengths of the LL field and the ID field are include in the length). For all user process data, the ID field is set to 0x12FF by LU 6.2. Other values of the ID field are used by IBM applications and by the LU for special purposes.

A logical record containing an ID field is also called a *GDS variable.* GDS is an acronym for Generalized Data Stream. A GDS variable can be comprised of multiple logical records, if the length of the MCR exceeds 32,763 bytes. But the utility of writing such large records is questionable, so we won't confuse the diagrams to show this general case.†

No matter how the data was written, using either the basic conversation write or the mapped conversation write, the LU is free to break up the logical records into *response units (RUs)* as it wishes. The maximum response unit size is negotiated by the two partner LUs when the session is started and its value is typically 256 bytes for an SDLC data link. The LU appends the *request header (RH)* to the response unit and passes the buffer to path control, as shown earlier.

† Indeed, a conforming LU 6.2 implementation need handle only 2048-byte logical records.

Confirmation

One parameter for the ALLOCATE verbs is the synchronization level. This can be set to NONE or CONFIRM.† Specifying the confirmation option allows the user processes to use the CONFIRM and CONFIRMED verbs, which were described earlier. This provides an end-to-end confirmation that is built into the LU 6.2 protocol itself. Note that the same type of feature can be built into any protocol, if the two user processes agree on a convention for indicating a confirmation. We'll see in Section 13.2 how the 4.3BSD line printer spooler uses a single byte message to provide a confirmation message to its partner process.

Buffering and Out-of-Band Data

LU 6.2 internally buffers almost everything that one process sends to another. To circumvent this, the FLUSH verbs are provided to force the LU to send what is has accumulated in its buffer. For example, when a program ALLOCATEs a conversation with another program and then executes a SEND_DATA to transfer data to that process, the LU, by default, won't send the allocate until its buffer is full. This means that the initiating process won't know if there was an error invoking the partner program until some number of SEND_DATA verbs cause the allocate to be sent to the other side.

LU 6.2 has no out-of-band data mechanism, comparable to what we described with TCP and SPP. A user process must issue a SEND_ERROR verb to notify the other process that attention is required. It is then up to the user processes to determine how to handle things.

Well-Known Addresses

If the first character of the remote transaction program name (TP name) in an ALLOCATE is a special character (0x00–0x0D or 0x10–0x3F), then the remote TP is a *service transaction program.* These programs are typically supplied by IBM and only privileged processes can allocate conversations with them. Three service TPs are DIA, DDM, and SNADS, described in the next section. This feature corresponds to the privileged port feature of 4.3BSD for ports in the Internet domain and in the XNS domain.

† A third option, SYNCPT is specified in the LU 6.2 protocol definition. This option allows conversations to specify synchronization points for potentially backing out changes that it has made. This option appears to have been discarded by IBM [IBM 1988]. This feature is similar to the Commitment, Concurrency, and Recovery feature of the OSI Common Application Service Element, which we describe in Section 5.6.7.

Option Sets

The LU 6.2 specification provides a base set of features that any implementation must provide, along with 41 optional features that can also be provided by a given product. For example, one option allows conversations between programs at the same LU. Another option allows a process to flush the LU's buffer using the `FLUSH` verbs.

Unfortunately, this laxness in the "official" specification of the protocol makes portability of LU 6.2 applications less than desired. But most real-world implementations of LU 6.2 on computer systems of general interest (i.e., ignoring special purpose systems that weren't intended to be programmed by end users) tend to support options that should have been in the base set. Furthermore, the publication by IBM of their CPI-Communications API [IBM 1988] gives some indication about the features in LU 6.2 which will be supported in the future.

5.4.6 Application Layer

IBM is building distributed applications using LU 6.2. IBM differentiates between two types of user processes that are at the layer above the LU:

- application transaction programs,
- service transaction programs.

The only real difference is that service TPs are supplied by IBM and a service TP can provide a service for an application TP. Application TPs are user processes. Service TPs can only be invoked by privileged processes, as described above. From a network programming perspective the difference is negligible. Both types of TPs access the network in the same way through the LU.

DIA—Document Interchange Architecture

DIA defines the functions required for interchanging documents between different IBM office systems. DIA is implemented on top of LU 6.2. That is, DIA uses LU 6.2 for communicating between different computer systems. DIA provides the following functions: document library services (storing and retrieving documents), document distribution (delivering documents to others in the network), and file transfer. DIA specifies the protocols and data structures used to exchange information between two DIA processes—a DIA client process and a DIA server process. An implementation of DIA usually provides a command language for an interactive user to execute DIA commands. These user commands (such as sign on, sign off, search, retrieve, deliver, delete, etc.) are translated into the appropriate DIA functions that are exchanged between the user's DIA client process and the DIA server process, using LU 6.2. *DISOSS* (Distributed Office Support System) is the IBM mainframe implementation of DIA that runs under CICS.

One type of document that is exchanged using DIA is one whose format is specified by DCA. DCA (Document Content Architecture) specifies the internal format of a document that can be exchanged between IBM office systems—the data stream and how it is to be interpreted. There are two forms of DCA, revisable form and final form. The revisable form specifies the structure of the document while it can be edited or formatted. The final form specifies the structure of the completed document in a device independent format. The intent of DCA is to allow a document to be moved between different IBM office systems, in either a revisable form or a final format.

SNADS—SNA Distribution Services

SNADS is an *asynchronous* distribution service for moving distribution objects (such as a document or an electronic mail message) from a source node to a destination node. By asynchronous we mean that it initiates the sending of the object, but the actual delivery might take place later. Furthermore, the delivery might require temporary storage in one or more intermediate nodes. The asynchronous nature of SNADS is similar to the Unix electronic mail delivery service provided by the Unix `uucp` program, which we describe in Section 5.7.

Contrast this with a *synchronous* delivery service where the source and destination are connected together through an LU 6.2 conversation. This synchronous capability is similar to the TCP/IP SMTP application, which uses a TCP connection between the source host and the destination host.

DDM—Distributed Data Management

DDM is another application available from IBM that uses LU 6.2. DDM is IBM's architecture for transparent remote file access in an SNA network. DDM allows an application program to access the records in a file on another system. To do this, the application program calls the local DDM interface to request a record from a file. If the interface recognizes the request as one for a local file, the local file is accessed. Otherwise, the local interface (the DDM client process) establishes an LU 6.2 session with the DDM server process on the remote system, and the server accesses the record and returns it to the client.

5.5 NetBIOS

5.5.1 History

In 1984 IBM released its first LAN, the IBM PC Network. It was similar in concept to an Ethernet, but ran at 2 Mbps, whereas most Ethernets operate at 10 Mbps. The interface card for the IBM PC (called an "adapter card" by IBM) was developed by Sytek, Inc., and contained on it the first implementation of NetBIOS. The name *NetBIOS* is derived from the name BIOS for the "basic input output system" for the IBM PC. The BIOS

was contained in read-only memory on the PC and provided an interface between a program on the PC and the actual hardware. Similarly, NetBIOS provides an interface between a program and the actual hardware on the interface card.

When IBM introduced its token ring LAN in 1985, it provided an implementation of NetBIOS for the token ring. The original PC Network implementation of NetBIOS was implemented in read-only memory on the interface card, while the token ring version was a software module. Despite the implementation differences, the token ring version provided the same interface to an application program as was provided by the original PC network.

The third implementation of NetBIOS by IBM occurred when the IBM PS/2 systems were introduced and the IBM LAN Support Program was available. This software package consists of device drivers and interface support for all of IBM's LAN interfaces.

NetBIOS is a software interface, not a network protocol. For example, the data packets that are exchanged across the IBM PC Network differ from those on a token ring network. We expect the actual frame that is transmitted by two different data-link layers to be different, since the data link header and trailer are different for each type of data link (token ring, Ethernet, SDLC, etc.). But the packet that is passed to the data-link layer should not change for a given protocol. For example, in the TCP/IP protocol suite, the IP datagram that starts with the IP header is the same, regardless of the data link being used. With NetBIOS this packet equivalency is not true. Nevertheless, the interfaces provided by all IBM implementations of NetBIOS are equivalent, providing a consistent software interface that has become a de facto standard for personal computers. In addition, there exist implementations of NetBIOS that use TCP and UDP as the underlying transport protocols, and standards exist for this in the Internet (RFC 1001 and RFC 1002).

Even though NetBIOS is not a protocol, we give an overview of the services that it provides here. Schwaderer [1988] provides details on using NetBIOS under the MS-DOS system. IBM [1987] provides additional details.

5.5.2 Overview

NetBIOS was designed for a group of personal computers, all sharing a common broadcast medium (the IBM PC Network, which we said earlier is like an Ethernet). It provides both a connection-oriented service (virtual circuit) and a connectionless (datagram) service. It supports both broadcast and multicast.

Four types of service are provided by NetBIOS:

- name service,
- session service,
- datagram service,
- general commands.

Figure 5.27 shows the relationship of these four services. Note that we do not indicate how these four services interact. In most implementations, a single box providing some form of datagram delivery (similar to the IP layer in the TCP/IP suite) is probably used.

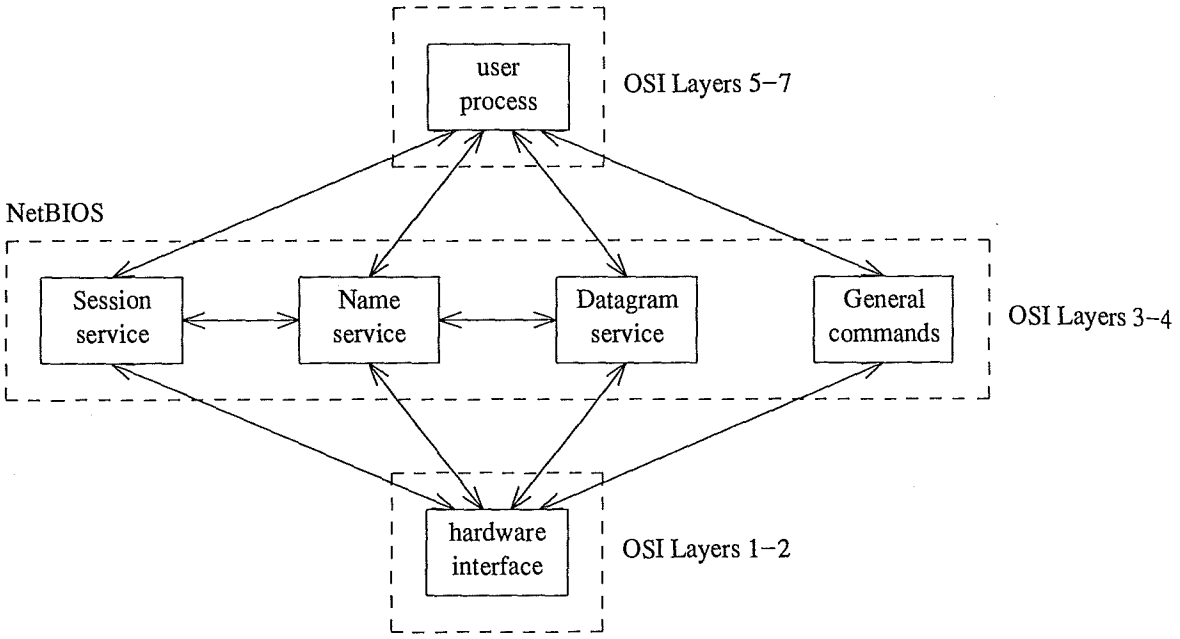


Figure 5.27 Relationship of NetBIOS services.

But unlike the IP layer, with NetBIOS the user process has no access to any services other than the ones described in the following sections, so the actual implementation need not concern us.

In many PC environments the application that NetBIOS is being used for is file sharing. In this case, another protocol interface exists above NetBIOS. This interface is called the Server Message Block protocol (SMB) and we show it in Figure 5.28.

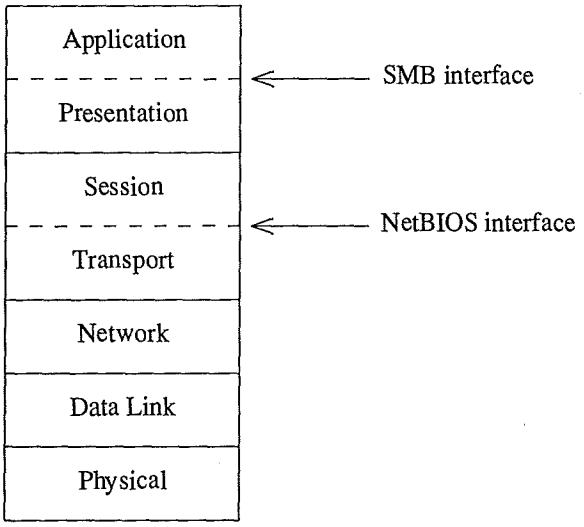


Figure 5.28 Relationship of NetBIOS and SMB to OSI model.

Dunsmuir [1989] shows the format of the SMB header and gives a listing of all the SMB

functions: create directory, open file, read from file, and so on. Line printer access is automatically handled by this software interface by providing three SMB operations that are for print files, not regular files. These open, write, and close a print spool file.

5.5.3 Name Service

Names are used to identify resources in NetBIOS. For example, for two processes to participate in a conversation, each must have a name. The client process identifies the specific server by the server's name, and the server can determine the name of the client. The name space is flat (that is, it is not hierarchical) and each name consists of from 1 to 16 alphanumeric characters. Upper case is different from lower case and names cannot start with an asterisk or with the three characters "IBM."

There are two types of names: *unique names* and *group names*. A unique name must be unique across the network. (As stated earlier, NetBIOS was designed for a LAN, so the name must typically be unique on the local network.) A group name does not have to be unique and all processes that have a given group name belong to the group.

There are four commands pertaining to name service.

`ADD_NAME`	Add a unique name
`ADD_GROUP_NAME`	Add a group name
`DELETE_NAME`	Delete name
`FIND_NAME`	Determine if a name is registered

To obtain a unique name or a group name, a process must bid for the use of the name. This is done by broadcasting a notice that the process wants to use the name, as either a unique name or a group name—by issuing either the `ADD_NAME` command or the `ADD_GROUP_NAME` command, respectively. If no objections are received from any other NetBIOS node, the name is considered registered by the requesting node. An objection occurs for an `ADD_NAME` if some other node responds that the name is currently in use on that node as either a unique name or a group name. The only restriction on the `ADD_GROUP_NAME` command is that no other node can be using the name as a unique name.

Implied in the NetBIOS specification is that each NetBIOS node maintains a table of all names that processes on that node currently own. These names continue to be owned by the NetBIOS node, until the names are specifically deleted, or until the node is powered off or reset. Realize that names are handled by two different entities: name registration is done by NetBIOS for a process that issues a `ADD_NAME` command, but it is NetBIOS that maintains the name table. Even though the process that registered the name might cease to exist, unless the name is specifically deleted with the `DELETE_NAME` command, the node's NetBIOS name table continues to know the name.

Both the `ADD_NAME` and `ADD_GROUP_NAME` commands return a *local name number* that is a small integer identifying the name. This number is used for the datagram commands and for the `RECEIVE_ANY` command (both described below).

By default, the IBM PC LAN Support Program transmits a name registration request

six times, at half-second intervals, until it considers the name registered by itself. This implies a 3-second delay every time an application that requires a new name is started for the first time.

The FIND_NAME command was added with the token ring implementation of NetBIOS and determines if a particular name has been registered by any other NetBIOS node.

5.5.4 Session Service

The NetBIOS session service provides a connection-oriented, reliable, full-duplex message service to a user process. The data is organized into messages and each message can be between 0 and 131,071 bytes. NetBIOS does not provide any form of out-of-band data.

The following commands provide session service:

CALL	Call—active open
LISTEN	Listen—passive open
SEND	Send session data
SEND_NO_ACK	Send session data, no acknowledgment
RECEIVE	Receive session data
RECEIVE_ANY	Receive session data
HANG_UP	Terminate session
SESSION_STATUS	Retrieve session status

NetBIOS requires one process to be the client and another to be the server. The server first issues a passive open with the LISTEN command. The client then connects with the server when the client executes the CALL command.

The LISTEN command requires the caller (typically a server process) to specify both the local name (which must be in the local NetBIOS name table) and the remote name. The local name is the *well-known address* that the server is known by. The remote name is the name of the specific client with which a session can be established. The caller can specify the remote name as an asterisk, allowing any remote process that specifies the local name to establish a connection. Since most servers are willing to accept a connection from any process (perhaps doing some form of authentication once the session is established), specifying the remote name as an asterisk is the typical scenario. Both ends of the session can access the name of the other end.

Both the LISTEN command and the CALL command return a *local session number* to the calling process. This small integer is then used for SEND and RECEIVE commands to specify a particular session (since a process can have more than one session active at any time). The local session number is also used by the HANG_UP command to specify which session is to be terminated. When a session is terminated, all pending data is first transferred.

In the normal SEND operation, the NetBIOS module waits for a positive acknowledgment from the other system before returning to the caller. Similarly, the RECEIVE command sends an acknowledgment to the other system before passing the received data

to the caller. This provides an end-to-end verification that the data was received. When the token ring implementation of NetBIOS appeared, IBM introduced the SEND_NO_ACK command, which does not perform the acknowledgments between the NetBIOS modules. The reason for this is that the data-link layer of the token ring network performs acknowledgments between the two data-link layers.

The IBM PC implementations provide a CHAIN_SEND command that combines two user buffers into a single message. The reason for this command is that the count associated with a normal SEND command is a 16-bit integer, which allows values between 0 and 65535. By combining two sends into a single operation, messages up to 131,071 bytes can be exchanged. This command, however, is an interface issue between the NetBIOS implementation and the user process, and we won't consider it any more in this text.

The RECEIVE command receives data for a particular session that the process has open. The local session number that was returned by the CALL or LISTEN specifies the session. The RECEIVE_ANY command allows a process to receive the next message from any of its current session partners. NetBIOS returns the actual session number corresponding to the received data.

5.5.5 Datagram Service

NetBIOS supports datagrams up to 512 bytes in length.† Datagrams can be sent to a specific name (either a unique name or a group name) or can be broadcast to the entire local area network. As with other datagram services, such as UDP/IP, the NetBIOS datagrams are connectionless and unreliable.

There are four datagram commands.

SEND_DATAGRAM	Send datagram
SEND_BROADCAST_DATAGRAM	Send broadcast datagram
RECEIVE_DATAGRAM	Receive datagram
RECEIVE_BROADCAST_DATAGRAM	Receive broadcast datagram

The SEND_DATAGRAM command requires the caller to specify the name of the destination. The name can be either a unique name or a group name. If the destination is a group name, then every member of the group receives the datagram. The caller of the RECEIVE_DATAGRAM command must specify the local name for which it wants to receive datagrams. Datagrams addressed to this name are received by the caller. The RECEIVE_DATAGRAM command also returns the name of the sender, in addition to the actual datagram data. If NetBIOS receives a datagram (i.e., the NetBIOS module has previously registered the name to which the datagram was addressed), but there are no RECEIVE_DATAGRAM commands pending, then the datagram is discarded.

The SEND_BROADCAST_DATAGRAM command sends the message to every

† The NetBIOS device driver supplied with the PC LAN Support Program allows a configuration option to specify a maximum datagram length other than 512 bytes.

NetBIOS system on the local network. When a broadcast datagram is received by a NetBIOS node, every process that has issued a RECEIVE_BROADCAST_DATAGRAM command receives the datagram. If none of these commands are outstanding when the broadcast datagram is received, the datagram is discarded. As with a normal datagram, the name of the broadcast datagram source is also returned to the receiver.

5.5.6 General Commands

There are four general commands.

RESET	Reset NetBIOS
CANCEL	Cancel an asynchronous command
ADAPTER_STATUS	Fetch adapter status
UNLINK	Unlink from bootstrap server

The RESET command clears the NetBIOS name and session tables, and also aborts any existing sessions.

The CANCEL command assumes that NetBIOS commands can be issued *asynchronously* by a user process. That is, the user process starts a command but does not wait for it to complete. Some method is required for the process to be notified when the command completes or for the process to check if a specific command is done or not. If asynchronous commands are supported by the NetBIOS implementation, then the CANCEL command cancels a specific outstanding command. If the command being cancelled is a SEND, then the associated session is aborted.

The STATUS command returns interface-specific status associated with either a local name or a remote name. Additionally, it returns the NetBIOS name table for that NetBIOS node (either the local node or a remote node). Unfortunately, for the IBM PC implementations of NetBIOS, the actual contents of the adapter status information that is returned to the caller depends on the adapter type (PC Network or token ring).

The UNLINK command was used with the original PC Network interface when a diskless workstation was bootstrapped from a remote disk drive.

5.5.7 NetBIOS Summary

In Figure 5.29 we add the two NetBIOS services to the table that we have been building. We use the terms **NbS** for the NetBIOS session services and **NbD** for the datagram services. Note that the NetBIOS session service is like LU 6.2—both assume the underlying data link provides reliability.

	IP	IDP	UDP	PEX	NbD	TCP	SPP	LU6.2	NbS
connection-oriented ?	no	no	no	no	no	yes	yes	yes	yes
message boundaries ?	yes	yes	yes	yes	yes	no	yes	yes	yes
data checksum ?	no	yes	opt.	yes	no	yes	yes	no	no
positive ack. ?	no	no	no	no	no	yes	yes	yes	yes
timeout and rexmit ?	no	no	no	yes	no	yes	yes	yes	yes
duplicate detection ?	no	no	no	no	no	yes	yes	yes	yes
sequencing ?	no	no	no	no	no	yes	yes	yes	yes
flow control ?	no	no	no	no	no	yes	yes	yes	yes
full-duplex ?						yes	yes	no	yes

Figure 5.29 Comparison of protocol features: Internet, XNS, SNA, and NetBIOS.

Using our definition of an *association* from the previous chapter, for the NetBIOS session service it consists of

- the protocol (NetBIOS session service),
- the source name,
- the source session number,
- the destination name,
- the destination session number.

An example could be

```
{NbS, JOESXT, 4, PRINTER, 7}
```

For the NetBIOS datagram service, only the protocol and the names are required, as there is no concept of a session for a datagram. Note also that the name of a partner process implies both the host name and the server's name.

5.6 OSI Protocols

5.6.1 Introduction

As stated in the previous chapter, the OSI model provides a framework within which standards can be developed for protocols at each layer. We refer to the protocol standards developed by ISO and other related organizations (CCITT, for example) as *OSI protocols*. This is in contrast to other networking protocols, most of which predate the OSI model, which have been developed by organizations other than ISO or CCITT. TCP/IP, XNS, and SNA, for example, are protocol suites that are not based on ISO standards.

OSI protocols have become popular lately as many organizations (such as the U.S. Government) have stated their intentions to move towards networks based on ISO standards. Unfortunately, networks based on OSI protocols are still in their infancy. Working examples of the lower layers exist, but most of the standards at these lower layers (layers 1–3) were developed before the OSI model. Standards exist and are currently being developed for the upper layers and for specific applications (see the description of the ISODE package below). In the remainder of this section we describe the existing ISO standards at each layer. We refer to the actual ISO standard documents by their 4-digit number.

As we mentioned in Chapter 4, the ISO standards differentiate between the services provided by a layer (for the layer above it) and the actual protocol used to provide the service. The protocol definitions describe the actual formats of the protocol packets, the header fields, and the like. Most layers or applications have both a standard for the service that it provides, along with a standard for the protocol used to provide that service.

A nonproprietary implementation of many of the OSI protocols is available as the ISODE software package. This package runs under 4.2BSD, 4.3BSD, System V Releases 2 and 3, along with other variants of these Unix systems. Refer to the Bibliography or Appendix A of Rose [1990] for ordering information. Version 5.0 of this package provides support for the following features:

- Transport services: TP0 and TP4,
- Session services,
- Presentation services,
- Association control services,
- Remote operation services (similar to the RPC techniques described in Chapter 18),
- ASN.1 abstract syntax notation tools,
- FTAM (transfer of text and binary files, directory listings, file management),
- FTAM/FTP gateway,
- VT (virtual terminal: basic class with TELNET profile),
- Directory services.

Future releases might provide the OSI electronic mail service (X.400) and a gateway between this and the Internet SMTP application. All these features and acronyms are described in the following sections.

Many people predict a gradual shift from non-OSI protocols to the OSI protocols between 1990 and 1995. The next major release of 4.xBSD should have support for some of the OSI protocols. The implementation of the upper layers in this release should be based on the ISODE package.

One confusing feature of the ISO standards is that their terminology differs from existing networking terminology. For example, what we call a client and server are termed an initiator and responder, respectively, in the ISO-world. The concepts of an iterative server and a concurrent server, which we introduced in Section 1.6, are called a static responder and a dynamic responder. The packets or messages that are exchanged by peer layers (see Figure 4.13, for example) are termed protocol data units in the OSI model.

For additional information on the OSI protocols, refer to Rose [1990], Henshall and Shaw [1988], Stallings [1987a], and Knightson, Knowles, and Larmouth [1988]. A detailed listing of all relevant ISO standards is in Chapin [1989].

5.6.2 Data-Link Layer

The data-link layer provides services to the network layer. LANs that use the OSI protocols typically use the IEEE 802 standards for the data-link layer and the physical layer. This provides for the IEEE 802.2 logical link control as the interface between the network layer and the data-link layer. The lower portion of the data-link layer, along with the physical layer, is then Ethernet (802.3), token bus (802.4), or token ring (802.5). These four IEEE standards have comparable ISO standards: 8802/2, 8802/3, 8802/4, and 8802/5. The 802.2 standard allows either a connection-oriented service or a connectionless service to be provided to the network layer. Stallings [1987b] provides additional details on the IEEE 802 standards.

Networks that use the OSI protocols with point-to-point connections typically use the link access procedure (LAP) that is part of the X.25 standard. (We mention X.25 below when we describe the network layer.) This protocol is similar to the SDLC protocol used by SNA for point-to-point links.

5.6.3 Network Layer

ISO standard 8348 defines the services provided by the network layer for the presentation layer. The original version of the standard provided only for a connection-oriented network service (CONS). An addendum provides for a connectionless network service (CLNS) also.

X.25 is the name used to describe the widely used connection-oriented protocol network layer protocol. X.25 is a CCITT standard that first appeared in 1974. X.25 encompasses layers 1, 2, and 3, not just the network layer. ISO standard 8878 describes how X.25 can be used to provide a connection-oriented network service.

ISO standard 8473 defines the protocol used to provide the connectionless network service. This protocol is similar to the Internet Protocol, IP, which we discussed in Section 5.2.4. One difference is that the Internet IP uses fixed-length address fields in its IP header (the 32-bit network ID and host ID value) while the OSI IP uses variable-length address fields.

5.6.4 Transport Layer

The task of the transport layer is to provide reliable, end-to-end data transfer for users of the transport layer. ISO standard 8072 provides the definition of the services provided by the transport layer. As with the network layer, the original standard only defined the services for a connection-oriented transmission, with an addendum specifying the services for connectionless transmission.

One service that the connection-oriented transport layer must provide is expedited

data, which we called out-of-band data in Chapter 4. Few specifics are given, however, other than the requirement that up to 16 bytes of expedited data be sent in a single operation. Additionally, the service definition requires that normal data sent after expedited data must not be delivered to the peer before the expedited data.

The definition of the transport layer services also includes features such as establishing a connection between two endpoints, and the negotiation of parameters during connection establishment.

The specification of the actual connection-oriented transport layer protocols is given in ISO standard 8073. Included in this standard is the definition of three different types of network services that are provided to the transport layer, types A, B, and C.

Type A A reliable network service. The network layer and the data-link layer handle all error conditions.

Type B A reliable network service with error notification. Although most error conditions are handled by the network layer and the data-link layer for this type of service, there can be some notifications to the transport layer that something has gone wrong. A reset notification from the network layer requires that both transport ends resynchronize. A restart notification requires that both transport ends establish a new connection.

Type C An unreliable network service. This is the type of service provided by datagram-oriented networks.

X.25 networks provide a type B network service, since both resets and restarts are possible. But, it is often assumed that an X.25 network provides a reliable type A service.

Given these three types of network services, there are five different classes of connection-oriented transport protocols: classes 0 through 4. We can classify the five protocol classes by the type of network service they are intended to be used with (A, B, or C), whether they can detect errors on their own, whether they can recover from errors that are signaled by the network layer, and whether they do multiplexing. We show this in Figure 5.30. Multiplexing here means the ability to have two or more transport connections over a single network connection.

Transport protocol class	Network service type	Error detection ?	Error recovery ?	Multiplexing ?
0	A	no	no	no
1	B	no	yes	no
2	A	no	no	yes
3	B	no	yes	yes
4	C	yes	yes	yes

Figure 5.30 ISO connection-oriented transport protocol classes.

(Figure 5.30 is a simplified description of these five classes. We could show over 20

different criteria, such as flow control, multiplexing, retransmission and timeout, and then note whether each of the five classes supports that feature.) These five classes are sometimes called TP0 through TP4. TP0 is a simple protocol—everything is handled by the lower layers. TP1 can be used with an X.25 network service, although if a reliable X.25 service is assumed, TP0 can be used instead. TP4 is similar to the Internet TCP, since TP4 assumes an unreliable network layer. TP4 could be used with the ISO connectionless network layer.

The ISO connectionless transport protocol is similar to UDP. It is defined in ISO standard 8602. A single packet format is defined and each packet contains the source address, destination address, an optional checksum, and the user data. The addresses can be used to identify user processes, similar to the port numbers used by UDP. Similar to UDP, if a checksum error is detected by the receiving transport layer, the packet is discarded.

5.6.5 Session Layer

The session layer provides services to a user process, in addition to the services provided by the transport layer. ISO standard 8326 defines the services to be provided by the session layer and ISO standard 8327 defines the session layer protocol.

Two of the services provided by the session layer to the layers above it are session establishment and session release. A session is similar in concept to a transport connection. During the life of a session there are two possible ways for the session layer to handle the transport connection that it needs for the session: a single transport connection can be used for the entire session, or two or more transport connections can be used for the entire session. In the latter case, it must be transparent to the user of the session layer that the actual transport connection has changed. It is also possible for a session layer to have consecutive sessions use a single transport connection. One restriction, however, is that the session layer cannot multiplex several sessions on a single transport connection.

Another service that can be provided by the session layer is dialog management. This feature provides a half-duplex, flip-flop form of data exchange. To manage this feature, an imaginary token is maintained by the two session layers. Only the end that holds the token can transmit data. During the session establishment, it is determined which end gets the token to start. One end can also ask the other end for the token when it wants to transmit data. This half-duplex, flip-flop mode of operation is similar to the SNA LU 6.2 protocol which we described in Section 5.4.5.

There are other services that the session layer can provide: synchronization, activity management, and exception reporting. Furthermore, the ISO standard defines four subsets of the session services, realizing that few applications, if any, need all the features that the session layer can provide. These four subsets are called kernel, BCS (basic combined subset), BSS (basic synchronized subset), and BAS (basic activity subset). The simplest of these, the kernel, must be provided with any implementation. All the kernel subset provides is session establishment and data transfer.

There is nothing similar to the session layer in the TCP/IP protocol suite.

5.6.6 Presentation Layer

The presentation layer is concerned with the representation of the data that is being exchanged. This can include conversion of the data between different formats (ASCII, EBCDIC, binary), data compression, and encryption. Additionally, the presentation layer must make the services of the session layer available to the application. Much of the presentation layer, therefore, is just a pass-through of application requests (establish a session, terminate a session, etc.) to the session layer.

ISO standard 8822 defines the services for the presentation layer and ISO standard 8823 defines the protocols.

One task of the presentation layer is to convert the application data into some standard form. To explain this we'll introduce the ISO terminology of *abstract syntax* and *transfer syntax*. The application layer deals with an abstract syntax. This includes items such as "an integer whose value is 1." This is an abstract description that does not say how the data value is represented. A transfer syntax, however, specifies exactly how this data value is represented. For example, it could be represented as 16-bit integer in twos complement binary format with the most significant bit transferred first. To convert from an abstract syntax to a transfer syntax, *encoding rules* are applied by the presentation layer. Two presentation layers exchange data in the transfer format, while the two application layers exchange data in the abstract format.

ISO standard 8824 specifies an abstract syntax called ASN.1. This stands for "abstract syntax notation 1." The encoding rules for converting ASN.1 data structures into a bit stream for transmission are contained in ISO standard 8825. A full description of ASN.1 is beyond the scope of this text. Refer to Section 8.2 of Tanenbaum [1989] for additional details. We'll discuss some other standards for data representation in Section 18.2 when we discuss remote procedure calls.

SNA supports some features that resemble the presentation layer. For example, the mapping of user data into a mapped conversation record, which we described in Section 5.4.5, is similar in concept.

5.6.7 Application Layer

Common Application Service Elements

CASE stands for "common application service elements." It is intended to provide capabilities that are useful to a variety of applications. Currently there are only two CASEs.

- Association Control Service Elements (ACSE).
  This element allows the user process to establish and release associations with a peer. There is a one-to-one relationship between associations and presentation layer connections.

- Commitment, Concurrency, and Recovery (CCR).
  CCR provides atomic actions between application entities. An atomic action is a set of operations, with either all operations being done or none of the operations being done—there is no in-between. Atomic operations and the techniques used by CCR have been used by distributed database systems and transaction processing systems for many years. This feature is similar to the syncpoint feature that we mentioned with the LU 6.2 `ALLOCATE` verb in Section 5.4.5.

ISO standard 8649 defines the CASE services and ISO standard 8650 defines the CASE protocols.

Electronic Mail

In 1984 CCITT defined a set of protocols for what it calls MHS (message handling system). The CCITT recommendations are defined in their X.400-series. These were incorporated in the OSI model at the application layer where they are called MOTIS (message-oriented text interchange system). X.400 provides for more than simple text-oriented electronic mail. It provides for a variety of message types, including text, facsimile, and digitized voice, for example.

Electronic mail under Unix is usually divided into two pieces. The user agent (UA) is the program the user interacts with the interactive user to send or receive mail. The user agent then communicates with a message transfer agent (MTA) that delivers the mail. Typical user agents are `/usr/ucb/Mail` on 4.3BSD, `/bin/mail` and `mailx` on System V, and a variety of other programs. The typical message transfer agent on 4.3BSD is `sendmail`. X.400 is concerned with all aspects of message handling—the user agent and the message transfer agent.

Directory Services

Directory services (DS) are similar to a telephone book. It maps names of people and services into their corresponding attributes (addresses, etc.). It is intended that the directory services be usable by the message handling systems (MHS) and other OSI applications. Directory services are sometimes classified as "white pages" or "yellow pages," similar to a telephone book, depending whether you are searching for a name or a service.

ISO standard 9594 and the CCITT X.500 recommendation specify all the details of the OSI directory.

Virtual Terminal

The OSI virtual terminal (VT) allows various terminals to be used. The intent is to isolate applications from the differences in terminal characteristics. ISO standards 9040 and 9041 describe the virtual terminal services and protocols, respectively.

When a virtual terminal connection is started, the two peer entities negotiate the parameters of the terminal that can be supported. An example of the types of parameters that can be specified for the virtual terminal are: number of dimensions (two for a standard CRT, three for a bit-mapped display), maximum coordinate in each dimension, allowable character sets, and so on. Some example operations that can be done are: move cursor to absolute position, enter characters starting at current position, and erase this line from cursor to end.

The ISO virtual terminal can be used to provide a remote login client and server, similar to the Internet TELNET application.

File Transfer, Access, and Management

The OSI file transfer, access, and management application (FTAM) is built around the concept of a virtual filestore. This virtual filestore presents a standard interface to its users. It is up to the software to map this virtual filestore into the actual filesystem being used. ISO standard 8571 specifies the services and protocols used by FTAM.

5.7 UUCP—Unix-to-Unix Copy

UUCP is the generic name used to describe a set of programs that can be used to copy files between different systems and to execute commands on other systems. An early version was made available with Version 7 Unix in 1978 and was intended for communication between systems using dial-up telephone lines. Today there are two major flavors of UUCP in use, the version distributed with 4.3BSD, which is derived from the Version 7 software, and a version known as Honey DanBer UUCP. (This name is derived from the login names of its three authors, Peter Honeyman, David A. Nowitz, and Brian E. Redman.) The Honey DanBer version is distributed with System V Release 3 where it is officially called BNU—Basic Networking Utilities. All versions of UUCP communicate with each other, so for our purposes it doesn't matter which specific version we describe. Details on the 4.3BSD version of UUCP can be found in Chapter 16 of Nemeth, Synder, and Seebas [1989]. Redman [1989] gives details on the Honey DanBer version, along with a history of UUCP.

UUCP is a collection of programs. The four that we're interested in are as follows:

`uucp` This program can be invoked by users to copy a file from one system to another. We'll use the term `uucp` to refer to this specific program, and the term UUCP to refer to the collection of programs. `uucp` is patterned after the Unix `cp` command, which copies one or more files. A typical use of `uucp` is

```
uucp main.c apple!~uucp
```

This command says to copy the file `main.c` in the current directory to the login directory of `uucp` on the system named `apple`.

`uux` This program spools a command for execution on another system. Although a user can execute this command, frequently this command is generated automatically by the mail software or the news software. (For additional details on the Usenet news system, refer to Tanenbaum [1989].)

`uucico` This program is usually run as a daemon process to perform the actions that have been requested by previous `uucp` or `uux` commands. Most Unix systems invoke the `uucico` program automatically at various times of the day from the `/usr/lib/crontab` file. This is one fundamental feature of the UUCP system—it is a batch mode system. The `uucp` and `uux` commands just queue work, and the work is executed at some later time by the `uucico` process.

`uuxqt` Executes files that were generated by `uux`. Normally `uuxqt` is invoked by `uux` or it is spawned by `uucico` to process execution files that have been received from another system.

Other programs exist in a typical UUCP software package—a program to display the jobs queued for transmission, one to remove a job that's been queued, and so on.

Figure 5.31 shows the typical operation of UUCP. Note that it is the `uucico` processes that communicate with each other across the network. There is no capability for a user process to use the UUCP protocols to communicate with some other user process.

The original `uucico` process supported a single data-link protocol known as the 'g' protocol. This was developed for dial-up or hardwired terminal lines and typically uses 64-byte packets. Its throughput is limited to around 9000 baud, even with higher line speeds. The 4.3BSD version supports two additional protocols. The 't' protocol assumes that the communication channel is error-free and no checksums are used. This protocol is typically used with TCP links. The 'f' protocol is used for X.25 links and relies on the flow control provided by the data stream. The 'f' protocol also applies a checksum only to the entire file (not to each packet) and uses a 7-bit data path, instead of the usual 8-bit data path.

Our overview of UUCP is to show where it belongs in relation to the other protocols described in this chapter. Since there is no ability for user processes to communicate across the network using UUCP, we don't describe it further in this text. Nevertheless, UUCP is an important piece of the networking tools used by most Unix sites.

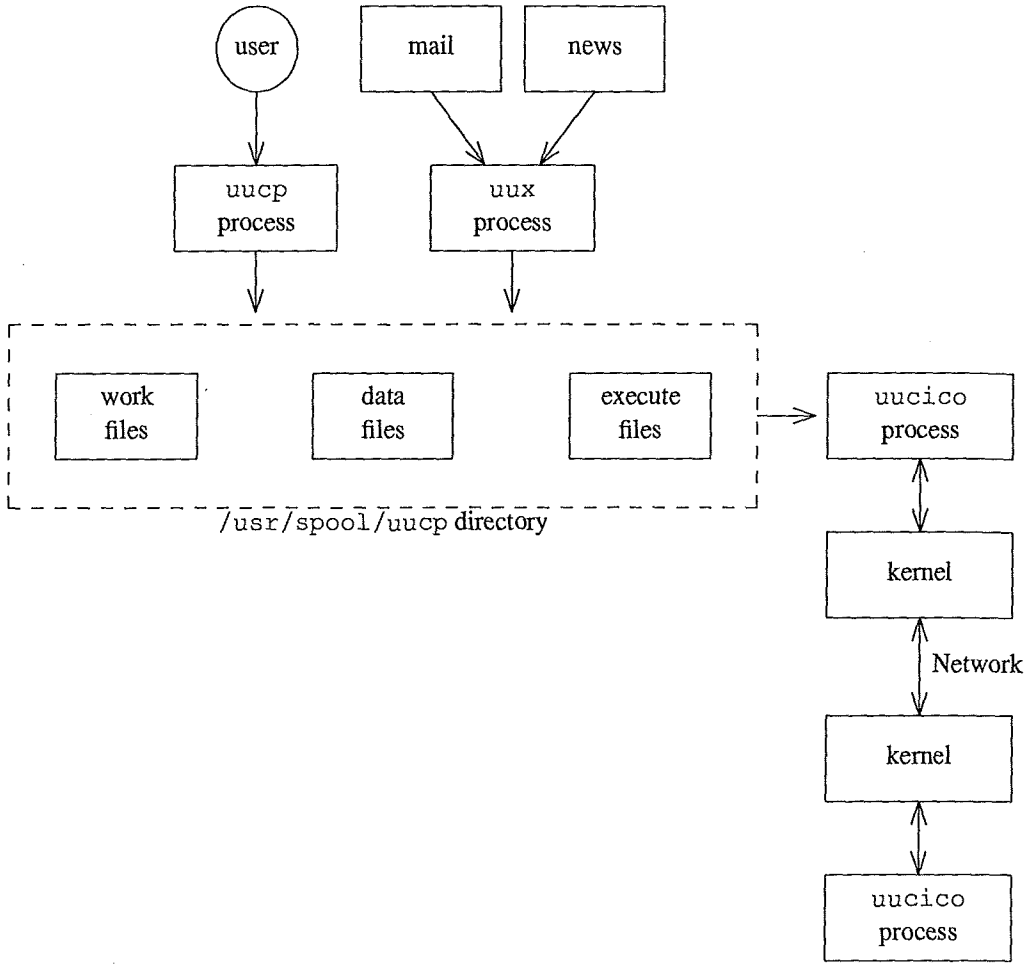


Figure 5.31 Overview of UUCP processes.

5.8 Protocol Comparisons

It is interesting to compare the types of services provided by the protocols we have discussed.

Fragmentation and Reassembly

TCP/UDP/IP — IP handles this. TCP and UDP need not worry about it, other than performance implications. Once a packet is fragmented, it is not reassembled until it reaches the destination host.

XNS — Never done. If a packet must be fragmented by some data-link layer, then it must be reassembled by the other end of the link, to prevent any other systems from knowing that it happened. The packet size is almost always 576 bytes, which is less than optimal for an Ethernet.

SNA	Done by path control.
NetBIOS	An implementation detail that is not specified.

Flow Control

IP	The IP layer sends ICMP source quench messages to a host when packets are arriving too quickly from that host. This is a form of hop-by-hop flow control.
TCP	Provides end-to-end flow control whereby the receiver tells the sender how much data it can accept.
UDP	No flow control is provided by UDP (other than the ICMP source quench feature of the underlying IP layer).
IDP	The error protocol can be used to notify a host when a packet from that host has been discarded because of resource limitations. This is similar to the ICMP source quench.
SPP	Provides end-to-end flow control, similar to TCP.
PEX	No flow control is provided (other than the error control packets provided by the underlying IDP layer).
SNA	Link-to-link flow control is provided by the data-link layer for most SNA data links (SDLC, token ring). Additionally, an end-to-end flow control is also provided by the LU.
NetBIOS	An implementation detail that is not specified.

Reliability

The issue of reliability is a touchy area when talking about whether some form of end-to-end reliability is needed. (This is one of the "religious topics" of networking.) Obviously, if the underlying data-link and network layers are unreliable, the transport layer must provide an end-to-end check. This is how TCP and UDP operate. But with UDP the end-to-end reliability is optional and there are some implementations that disable it when running on a "reliable" LAN such as an Ethernet.† The reason we say that an Ethernet is reliable is that every Ethernet frame has a 32-bit CRC appended to it, to verify that the data is exchanged without any errors between the two Ethernet interface cards. Other LAN technology, such as the token ring, also provides a 32-bit CRC to provide link-to-link reliability.

If the data-link layer is reliable, do we still need an end-to-end reliability check?

† If you're running some form of BSD Unix, and if you have adequate permission and knowledge, look at the global variable `udpcksum` in the kernel with a debugger. If its value is zero, UDP checksums are not calculated for outgoing UDP packets. A value of one enables the checksum calculation for outgoing packets.

Without answering this touchy question we should point out where the data could be corrupted, even if it is transferred between the two interface cards without any errors. First, the interface card contains memory in which the incoming and outgoing frames are stored. This memory can be a source of errors. The data is also transferred between the interface card and the computer's memory, typically along an input/output bus of some form. On some LANs there might also be bridges or repeaters, which can also be the source of errors.

IP	Provides a 16-bit checksum that covers the IP header only. The upper layers (TCP and UDP) are left to do their own checksum of the actual message, if desired.
TCP	Assumes the IP layer is unreliable and provides a 16-bit checksum of the entire message.
UDP	As with TCP, it provides a 16-bit checksum of the entire message.
XNS	IDP provides a 16-bit checksum of the entire packet, therefore no additional end-to-end reliability checks are required by the upper layers (SPP and PEX).
SNA	Assumes the data-link layer provides the reliability. No form of end-to-end checksum is provided.
NetBIOS	Both the datagram service and the session service assume the data link layer is reliable, with no additional checksums provided.

Sequencing

The datagram protocols do not perform sequencing—IP, UDP, IDP, PEX, and NetBIOS datagram. The connection-oriented protocols all provide this feature—TCP, SPP, LU 6.2, and NetBIOS session. SNA assumes that the data-link layer does the sequencing, with a verification of this taking place in the LU. NetBIOS also assumes the data link does the sequencing. TCP and SPP both do the sequencing themselves, assuming the underlying layers do not sequence the data.

Timeout and Retransmission

Most datagram protocols do not provide this service (IP, UDP, IDP, and NetBIOS). PEX, however, is a datagram service that provides a timeout and retransmission. As expected, the connection-oriented protocols do provide this service—TCP, SPP, LU 6.2, and NetBIOS. As with sequencing, both SNA and NetBIOS require the data link to handle this.

5.9 Summary

We have covered five protocol suites in this chapter.

- TCP/IP
- XNS
- SNA
- NetBIOS
- OSI
- UUCP

Our emphasis has been the TCP/IP protocol suite, as that is the one used for most of the examples in the later chapters. We'll also show examples using XNS in Chapters 6 and 11. Protocol comparisons are worthwhile, to help put things into perspective.

We've covered many protocols in this chapter, not all of which are essential for network programming. For example, although you probably won't use the ARP or RARP protocols directly (compared to UDP or TCP), you will encounter RARP as soon as you connect a diskless workstation to a TCP/IP internet.

In the next two chapters we'll cover two application program interfaces (APIs) between a user process and a networking protocol: Berkeley sockets and the System V Transport Layer Interface.

Exercises

5.1 What is the maximum number of hosts on a TCP/IP internet?

5.2 What is the maximum number of hosts on a XNS internet?

5.3 Given the addressing structure of TCP/IP and XNS, would it be feasible for every home in the U.S. to have a unique network address? How about every home in the world?

5.4 Compare the OSI connectionless network service (Section 5.6.3) with IP.

6

Berkeley Sockets

6.1 Introduction

This chapter describes the first of several *application program interfaces (APIs)* to the communication protocols. The API is the interface available to a programmer. The availability of an API depends on both the operating system being used, and the programming language. The two most prevalent communication APIs for Unix systems are Berkeley sockets and the System V Transport Layer Interface (TLI). Both of these interfaces were developed for the C language. This chapter describes the Berkeley socket interface, and the next chapter describes TLI.

First, let's compare network I/O to file I/O. For a Unix file, there are six system calls for input and output: `open`, `creat`, `close`, `read`, `write`, and `lseek`. All these system calls work with a file descriptor, as described in Section 2.3. It would be nice if the interface to the network facilities maintained the file descriptor semantics of the Unix filesystem, but network I/O involves more details and options than file I/O.† For example, the following points must be considered:

† The original TCP/IP implementation for the BSD version of Unix was developed by Bolt, Beranek, and Newman (BBN) under a DARPA contract in 1981 [Walsh and Gurwitz 1984]. It is interesting to note that this version used these six system calls for both file I/O and network I/O. The socket interface and the interprocess communication facilities, which we describe in this chapter, were then added by Berkeley with the 4.1cBSD release.

- The typical client–server relationship is not symmetrical. To initiate a network connection requires that the program know which role (client or server) it is to play.
- A network connection can be connection-oriented or connectionless. The former case is more like file I/O than the latter, since once we open a connection with another process, the network I/O on that connection is always with the same peer process. With a connectionless protocol, there is nothing like an "open" since every network I/O operation could be with a different process on a different host.
- Names are more important in networking than for file operations. For example, a program that is passed a file descriptor by a parent process can do file I/O on it without ever needing to know the original name that the file was opened under. The file descriptor is all the process needs. A networking application, however, might need to know the name of its peer process to verify that the process has authority to request the services.
- Recall our definition of an association in Chapter 4

  {protocol, local-addr, local-process, foreign-addr, foreign-process}

  There are more parameters that must be specified for a network connection, than for file I/O. Also, as we saw in Chapter 5, each of the parameters can differ from one protocol to the next. Compare, for example, the format of either the *local-addr* or the *foreign-addr* between the Internet and XNS: the Internet address is 4 bytes while the XNS address is 10 bytes.
- For some communication protocols, record boundaries have significance. The Unix I/O system is stream oriented, not message oriented, as discussed in Section 3.6.
- The network interface should support multiple communication protocols. For example, the network functions can't use a 32-bit integer to hold network addresses, since even though this is adequate for Internet addresses, it is inadequate for XNS addresses. Generic techniques must be used to handle features that can change from one protocol to another, such as addresses. Supporting multiple communication protocols is almost akin to having Unix support multiple file access techniques. Imagine the Unix I/O system if the kernel had to support Unix I/O along with three or four other techniques (DEC's RMS, IBM's VSAM, etc.).

As another comparison between network I/O and file I/O, Figure 6.1 shows some of the steps required to use sockets, TLI, System V message queues, and FIFOs, for a connection-oriented transfer. We'll cover the details in Figure 6.1 as we proceed through this chapter and the next. Our purpose now is to show the added complexity imposed by the networking routines, compared to message queues and FIFOs.

		Sockets	TLI	Messages	FIFOs
Server:	allocate space		`t_alloc()`		
	create endpoint	`socket()`	`t_open()`	`msgget()`	`mknod()` `open()`
	bind address	`bind()`	`t_bind()`		
	specify queue	`listen()`			
	wait for connection	`accept()`	`t_listen()`		

	get new fd		`t_open()`
			`t_bind()`
			`t_accept()`		

Client:	allocate space		`t_alloc()`		
	create endpoint	`socket()`	`t_open()`	`msgget()`	`open()`
	bind address	`bind()`	`t_bind()`		
	connect to server	`connect()`	`t_connect()`		

	transfer data	`read()`
		`write()`
		`recv()`
		`send()`
			`read()`
			`write()`
			`t_rcv()`
			`t_snd()`
				`msgrcv()`
				`msgsnd()`
					`read()`
					`write()`

	datagrams	`recvfrom()` `sendto()`	`t_rcvudata()` `t_sndudata()`		

	terminate	`close()`
		`shutdown()`
			`t_close()`
			`t_sndrel()`
			`t_snddis()`
				`msgctl()`	`close()`
					`unlink()`

Figure 6.1 Comparison of sockets, TLI, message queues, and FIFOs.

In the examples that we show in this chapter we must specify the type of process (client or server) and the type of protocol (connection-oriented or connectionless). Furthermore, for the server examples we have to specify if the server is a concurrent server or an iterative server. (Usually it doesn't matter to the client whether it is communicating with a concurrent server or an iterative server.) This gives four potential combinations.

	iterative server	concurrent server
connection-oriented protocol	infrequent (Daytime)	typical
connectionless protocol	typical	infrequent (TFTP)

The Internet Daytime protocol, which we describe in Section 10.2, is an example of a connection-oriented protocol whose server is usually implemented using an iterative server. Indeed, the 4.3BSD Internet superserver which we describe in Section 6.16 provides an iterative server for this protocol. We provide an example of a concurrent server using a connectionless protocol in Chapter 12—the TFTP server.

6.2 Overview

The socket interface was first provided with the 4.1cBSD system for the VAX, circa 1982. The interface that we describe here corresponds to the original 4.3BSD VAX release from 1986. This release supported the following communication protocols:

- Unix domain (described in Section 6.3)
- Internet domain (TCP/IP)
- Xerox NS domain

Be aware that some vendors who provide 4.3BSD-based systems do not provide the XNS support that was provided by Berkeley. Nevertheless, we show examples using XNS, since it provides a good working example, in addition to TCP/IP, for some of the portability problems in network programming.

Figure 6.2 shows a time line of the typical scenario that takes place for a connection-oriented transfer—first the server is started, then sometime later a client is started that connects to the server.

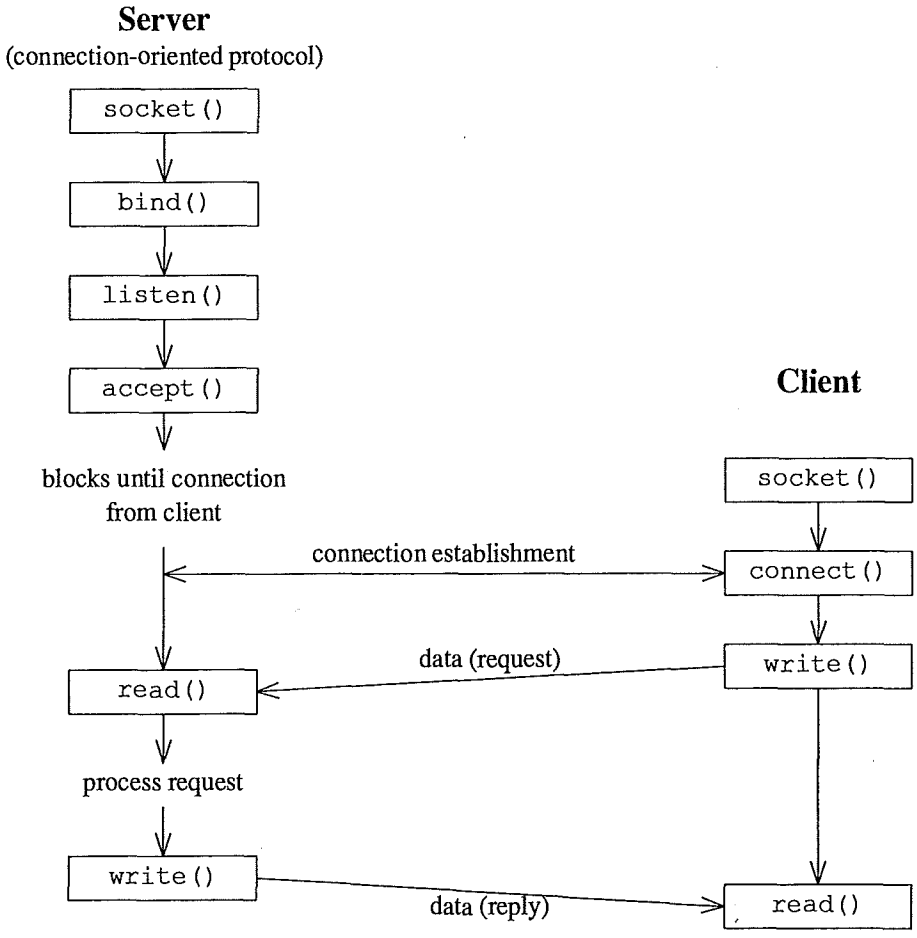


Figure 6.2 Socket system calls for connection-oriented protocol.

For a client–server using a connectionless protocol, the system calls are different. Figure 6.3 shows these system calls. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto` system call, which requires the address of the destination (the server) as a parameter. Similarly, the server does not have to accept a connection from a client. Instead, the server just issues a `recvfrom` system call that waits until data arrives from some client. The `recvfrom` returns the network address of the client process, along with the datagram, so the server can send its response to the correct process.

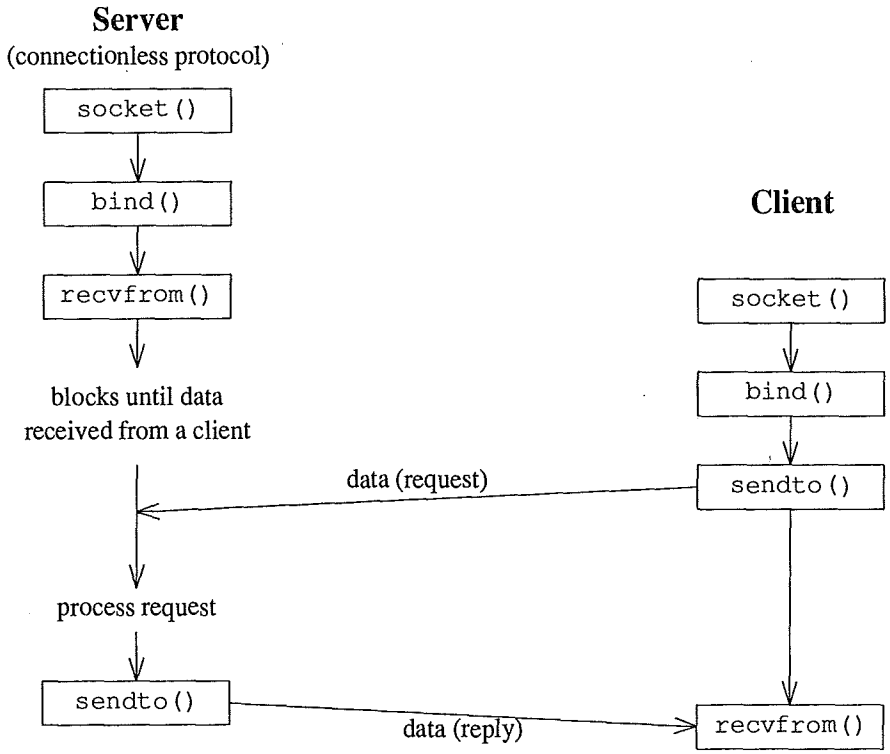


Figure 6.3 Socket system calls for connectionless protocol.

6.3 Unix Domain Protocols

4.3BSD provides support for what it calls the "Unix domain" protocols. But unlike other domains, such as the Internet domain and the XNS domain, sockets in the Unix domain can only be used to communicate with processes on the same Unix system. We did not provide a description of the Unix domain protocols in the previous chapter, as they are limited to 4.3BSD, and are not "communication protocols" in the true sense of the term. They are a form of IPC, but their implementation has been built into the 4.3BSD networking system in the same fashion as other true communication protocols.

4.3BSD provides both a connection-oriented interface and a connectionless interface to the Unix domain protocols. Both can be considered reliable, since they exist only within the kernel and are not transmitted across external facilities such as a communication line between systems. Checksums and the like are not needed. As with other connection-oriented protocols (TCP and SPP, for example), the connection-oriented Unix version provides flow control. In a similar fashion, as with other connectionless protocols (UDP and IDP, for example), the Unix domain datagram facility does *not* provide flow control. This has implications for user programs, since it is possible for a datagram client to send data so fast that buffer starvation can occur. If this happens, the sender must try to send the data repeatedly. For this reason alone, it is recommended that you use the connection-oriented Unix domain protocol.

The Unix domain protocols provide a feature that is not currently provided by any other protocol family: the ability to pass access rights from one process to another. We'll discuss this feature in more detail in Section 6.10 when we describe the passing of file descriptors between processes.

Unlike the other protocols we've covered, there is nothing like encapsulation performed on the Unix domain messages—their actual implementation is a kernel detail that need not concern us. As it turns out, 4.3BSD implements pipes using the connection-oriented Unix domain protocol.

The name space used by the Unix domain protocols consists of pathnames. A sample association could be

```
{unixstr, 0, /tmp/log.01528, 0, /dev/logfile}
```

The *protocol* here is `unixstr`, which stands for "Unix stream," the connection-oriented protocol. The *local-process* in this example is `/tmp/log.01528` and the *foreign-process* is `/dev/logfile`. We show both the *local-addr* and the *foreign-addr* as zero, since the pathnames on the local host are the only addresses used in this domain.† An association using the connectionless Unix protocol is similar, except that we use the term `unixdg` to specify the datagram protocol, the first member of the 5-tuple.

The 4.3BSD implementation creates a file in the filesystem with the specified pathname, although there are comments in the 4.3BSD manuals indicating that future versions might not create these files. This is somewhat misleading, however, because these filesystem entries are not true "files." For example, we cannot open these files with the `open` system call. These files have a type of `S_IFSOCK` as reported by the `stat` or `fstat` system calls.

† We could define the *local-addr* and the *foreign-addr* to be the pathnames, with the *local-process* and *foreign-process* both being zero. Our reason for specifying the association as we did is to reiterate that a host address is not required since the association is limited to processes on the local host.

6.4 Socket Addresses

Many of the BSD networking system calls require a pointer to a socket address structure as an argument. The definition of this structure is in <sys/socket.h>:

```
struct sockaddr {
  u_short  sa_family;    /* address family: AF_xxx value */
  char     sa_data[14];  /* up to 14 bytes of protocol-specific
                            address */
};
```

The contents of the 14 bytes of protocol-specific address are interpreted according to the type of address. For the Internet family, the following structures are defined in <netinet/in.h>:

```
struct in_addr {
  u_long  s_addr;              /* 32-bit netid/hostid */
                               /* network byte ordered */
};

struct sockaddr_in {
  short           sin_family;  /* AF_INET */
  u_short         sin_port;    /* 16-bit port number */
                               /* network byte ordered */
  struct in_addr  sin_addr;    /* 32-bit netid/hostid */
                               /* network byte ordered */
  char            sin_zero[8]; /* unused */
};
```

As we'll see throughout this chapter, almost every code example has

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

at the beginning. This header file provides C definitions and data type definitions (typedefs) that are used throughout the system. We will mainly use the names defined for the four unsigned integer datatypes, which we show in Figure 6.4. Unfortunately these four names differ between 4.3BSD and System V.

C Data type	4.3BSD	System V
unsigned char	u_char	unchar
unsigned short	u_short	ushort
unsigned int	u_int	uint
unsigned long	u_long	ulong

Figure 6.4 Unsigned data types defined in <sys/types.h>.

For the Xerox NS family, the following structures are defined in <netns/ns.h>:

```
union ns_host {
  u_char        c_host[6]; /* hostid addr as six bytes */
  u_short       s_host[3]; /* hostid addr as three 16-bit shorts */
                           /* network byte ordered */
};

union ns_net {
  u_char        c_net[4];  /* netid as four bytes */
  u_short       s_net[2];  /* netid as two 16-bit shorts */
                           /* network byte ordered */
};

struct ns_addr {    /* here is the combined 12-byte XNS address */
  union ns_net   x_net;    /* 4-byte netid */
  union ns_host  x_host;   /* 6-byte hostid */
  u_short        x_port;   /* 2-byte port (XNS "socket") */
                           /* network byte ordered */
};

struct sockaddr_ns {
  u_short         sns_family;  /* AF_NS */
  struct ns_addr  sns_addr;    /* the 12-byte XNS address */
  char            sns_zero[2]; /* unused */
};
#define  sns_port  sns_addr.x_port
```

Things are more complicated with XNS, as some of the network code wants to get at the hostid and netid fields in different ways.

For the Unix domain, the following structure is defined in <sys/un.h>:

```
struct sockaddr_un {
  short  sun_family;     /* AF_UNIX */
  char   sun_path[108];  /* pathname */
};
```

(You may have noticed the discrepancies in the declarations of the first two bytes of these address structures, the XX_family members. The Internet and Unix domain members are declared as short integers, while the XNS member and the generic sockaddr member are unsigned short integers. Fortunately the values stored in these variables, the AF_xxx constants that we define in the next section, all have values between 1 and 20, so this discrepancy in the data types doesn't matter.)

Unlike most Unix system calls, the BSD network system calls don't assume that the Unix pathname in sun_path is terminated with a null byte. Notice that this protocol-specific structure is larger than the generic sockaddr structure, while the sockaddr_in and sockaddr_ns structures were both identical in size (16 bytes) to

the generic structure. The system interface currently supports structures up to 110 bytes, but some of the generic system network routines (the routing tables in the kernel, for example) only support the 16-byte structures. The Unix domain protocols, however, can be larger than 16 bytes since they don't use these facilities. To handle socket address structures of different sizes, the system interface always passes the size of the address structure, in addition to a pointer to the structure, as we describe in the next paragraph.

A picture of these socket address structures is shown in Figure 6.5.

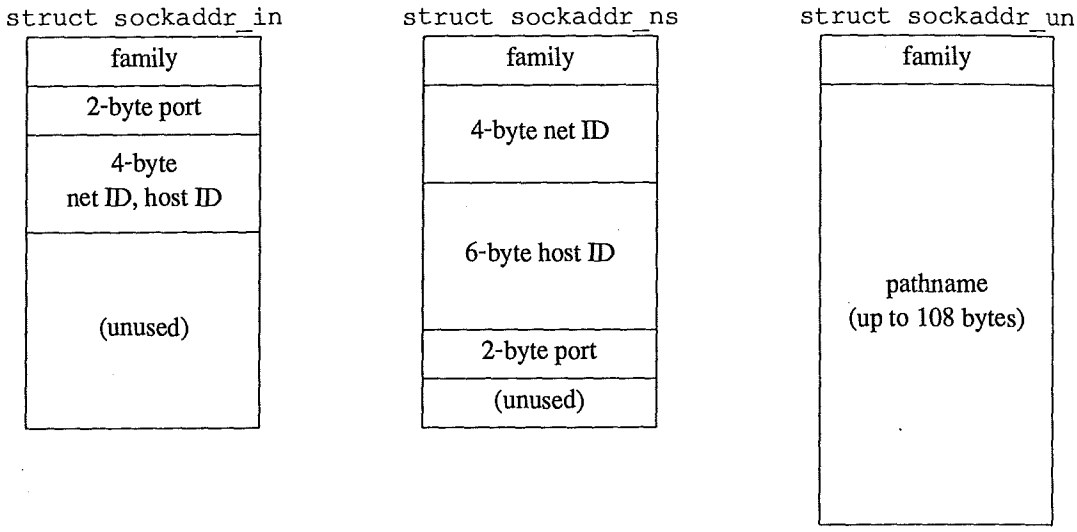


Figure 6.5 Socket address structures for Internet, XNS and Unix families.

The network system calls, such as `connect` and `bind`, work with any of the supported domains, so a technique must be used to pass any of the socket address structures shown in Figure 6.5, `sockaddr_in`, `sockaddr_ns`, or `sockaddr_un`, to these generic system calls. The network system calls take two arguments: the address of the generic `sockaddr` structure and the size of the actual protocol-specific structure. What the caller must do is provide the address of the protocol-specific structure as an argument, casting this pointer into a pointer to a generic `sockaddr` structure. For example, to invoke the `connect` system call for an Internet domain socket, we write

```
struct sockaddr_in  serv_addr; /* Internet-specific addr struct */
    ...
(fill in Internet-specific information)
    ...
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

The second argument is a pointer to the Internet address structure and the third argument is its size (16 bytes).

The designers of the socket interface could have also chosen to use a C union to define the socket address structure. The declaration could look like

```
struct sock_addr {
  short  sa_family;      /* AF_xxx value */
  union {
    struct sockaddr_in  sa_in;    /* Internet address */
    struct sockaddr_ns  sa_ns;    /* XNS address */
    struct sockaddr_un  sa_un;    /* Unix address */
  } sa_val;
};
```

The problem with this approach is that the size of the union is determined by the size of the largest member, which is the Unix domain address. This would cause every `sock_addr` structure to be 110 bytes in size, even though Internet and XNS addresses need only 8 bytes and 14 bytes, respectively. (Note that in this case the 8 bytes of zero at the end of an Internet `sockaddr_in` and the 2 bytes of zero at the end of an XNS `sockaddr_ns` are not needed.) One advantage of a union is that the size of the structure would not have to be an argument (and sometimes a result) to the system calls, as we'll see below. The actual interface design and the choice of a generic `sockaddr` structure along with the protocol-specific socket address structures, was a compromise.

6.5 Elementary Socket System Calls

We now describe the elementary system calls required to perform network programming. In the following section we use these system calls to develop some networking examples.

`socket` System Call

To do network I/O, the first thing a process must do is call the `socket` system call, specifying the type of communication protocol desired (Internet TCP, Internet UDP, XNS SPP, etc.).

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

int socket(int family, int type, int protocol);
```

The *family* is one of

`AF_UNIX`	Unix internal protocols
`AF_INET`	Internet protocols
`AF_NS`	Xerox NS protocols
`AF_IMPLINK`	IMP link layer

The `AF_` prefix stands for "address family." There is another set of terms that is defined, starting with a `PF_` prefix, which stands for "protocol family": `PF_UNIX`, `PF_INET`, `PF_NS`, and `PF_IMPLINK`. Either term for a given family can be used, as they are equivalent.

An IMP is an *Interface Message Processor*. This is an intelligent packet switching node that was referred to in our description of the ARPANET in the previous chapter. These nodes are connected with point-to-point links, typically leased telephone lines. The BSD system provides a raw socket interface to the IMP, which is what the AF_IMPLINK address family is used for. We won't concern ourselves further with this special purpose network interface.

The socket *type* is one of the following:

SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RDM	reliably delivered message socket (not implemented yet)

Not all combinations of socket *family* and *type* are valid. Figure 6.6 shows the valid combinations, along with the actual protocol that is selected by the pair.

	AF_UNIX	AF_INET	AF_NS
SOCK_STREAM	Yes	TCP	SPP
SOCK_DGRAM	Yes	UDP	IDP
SOCK_RAW		IP	Yes
SOCK_SEQPACKET			SPP

Figure 6.6 Protocols corresponding to socket *family* and *type*.

The boxes marked "Yes" are valid, but don't have handy acronyms. The empty boxes are not implemented.

The *protocol* argument to the socket system call is typically set to 0 for most user applications. There are specialized applications, however, that specify a *protocol* value, to use a specific protocol. The valid combinations are shown in Figure 6.7.

family	*type*	*protocol*	Actual protocol
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAW	IPPROTO_RAW	(raw)
AF_NS	SOCK_STREAM	NSPROTO_SPP	SPP
AF_NS	SOCK_SEQPACKET	NSPROTO_SPP	SPP
AF_NS	SOCK_RAW	NSPROTO_ERROR	Error protocol
AF_NS	SOCK_RAW	NSPROTO_RAW	(raw)

Figure 6.7 Combinations of *family*, *type*, and *protocol*.

The IPPROTO_xxx constants are defined in the file <netinet/in.h> and the NSPROTO_xxx constants are defined in the file <netns/ns.h>. In Section 11.2 we'll show the Internet ping program, which uses the ICMP protocol.

The socket system call returns a small integer value, similar to a file descriptor. We'll call this a socket descriptor, or a *sockfd*. To obtain this socket descriptor, all we've specified is an address family and the socket type (stream, datagram, etc.). For an association

{protocol, local-addr, local-process, foreign-addr, foreign-process}

all the socket system call specifies is one element of this 5-tuple, the *protocol*. Before the socket descriptor is of any real use, the remaining four elements of the association must be specified. To show what a typical process does next, we'll differentiate between both client and server, and between a connection-oriented protocol and a connectionless protocol. Figure 6.8 shows the typical system calls for each case.

	protocol	*local-addr, local-process*	*foreign-addr, foreign-process*
connection-oriented server	socket()	bind()	listen(), accept()
connection-oriented client	socket()	connect()	
connectionless server	socket()	bind()	recvfrom()
connectionless client	socket()	bind()	sendto()

Figure 6.8 Socket system calls and association elements.

socketpair System Call

This system call is implemented only for the Unix domain.

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

int socketpair(int family, int type, int protocol, int sockvec[2]);
```

This returns two socket descriptors, *sockvec[0]* and *sockvec[1]*, that are unnamed and connected. This system call is similar to the pipe system call, but socketpair returns a pair of socket descriptors, not file descriptors. Additionally, the two socket descriptors returned by socketpair are bidirectional, unlike pipes, which are unidirectional. Recalling our pipe examples from Section 3.4, we had to execute the pipe system call twice, obtaining four file descriptors, to get a bidirectional flow of data between two processes. With sockets this isn't required. We'll call these bidirectional, connection-oriented, Unix domain sockets *stream pipes*. We return to these stream pipes in Section 6.9.

Since this system call is limited to the Unix domain, there are only two allowable versions of it

```
int     rc, sockfd[2];

rc = socketpair(AF_UNIX, SOCK_STREAM, 0, sockfd);
```

or

```
rc = socketpair(AF_UNIX, SOCK_DGRAM, 0, sockfd);
```

`bind` System Call

The `bind` system call assigns a name to an unnamed socket.

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

int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

The second argument is a pointer to a protocol-specific address and the third argument is the size of this address structure. There are three uses of `bind`.

1. Servers register their well-known address with the system. It tells the system "this is my address and any messages received for this address are to be given to me." Both connection-oriented and connectionless servers need to do this before accepting client requests.
2. A client can register a specific address for itself.
3. A connectionless client needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to. This corresponds to making certain an envelope has a valid return address, if we expect to get a reply from the person we send the letter to.

The `bind` system call fills in the *local-addr* and *local-process* elements of the association 5-tuple.

`connect` System Call

A client process `connects` a socket descriptor following the `socket` system call to establish a connection with a server.

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

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

The *sockfd* is a socket descriptor that was returned by the `socket` system call. The second and third arguments are a pointer to a socket address, and its size, as described earlier.

For most connection-oriented protocols (TCP and SPP, for example), the `connect` system call results in the actual establishment of a connection between the local system and the foreign system. Messages are typically exchanged between the two systems and specific parameters relating to the conversation might be agreed on (buffer sizes, amount of data to exchange between acknowledgments, etc.). In these cases the `connect` system call does not return until the connection is established, or an error is returned to the process. Section 12.15 of Comer [1988] discusses the three-way handshake used by TCP to establish a connection. Section 7.4 of Xerox [1981b] discusses a similar technique used by SPP.

The client does not have to `bind` a local address before calling `connect`. The connection typically causes these four elements of the association 5-tuple to be assigned: *local-addr*, *local-process*, *foreign-addr*, and *foreign-process*. In all the connection-oriented client examples that we'll show, we'll let `connect` assign the local address. This is what we diagramed in Figure 6.2 and Figure 6.8.

A connectionless client can also use the `connect` system call, but the scenario is different from what we just described. For a connectionless protocol, all that is done by the `connect` system call is to store the *servaddr* specified by the process, so that the system knows where to send any future data that the process writes to the *sockfd* descriptor. Also, only datagrams from this address will be received by the socket. In this case the `connect` system call returns immediately and there is not an actual exchange of messages between the local system and the foreign system.

One advantage of `connecting` a socket associated with a connectionless protocol is that we don't need to specify the destination address for every datagram that we send. We can use the `read`, `write`, `recv`, and `send` system calls. (We describe the system calls used for socket I/O later in this section.)

There is another feature provided for connectionless clients that call `connect`. If the datagram protocol supports notification for invalid addresses, then the protocol routine can inform the user process if it sends a datagram to an invalid address. For example, the Internet protocols specify that a host should generate an ICMP port unreachable message if it receives a UDP datagram specifying a UDP port for which no process is waiting to read from. The host that sent the UDP datagram and receives this ICMP message can try to identify the process that sent the datagram, and notify the process. 4.3BSD, for example, notifies the process with a "connection refused" error (`ECONNREFUSED`) on the next system call for this socket, only if the process had `connected` the socket to the destination address. We'll see an example of this with our Internet time client in Section 10.2.

Note that the `connect` and `bind` system calls require only the pointer to the address structure and its size as arguments, not the protocol. This is because the protocol is available from two places: first, the `AF_xxx` value is always contained in the first two

bytes of the socket address structure; second, these system calls all require a socket descriptor as an argument, and this descriptor is always associated with a single protocol family, from the `socket` system call that created the descriptor.

`listen` System Call

This system call is used by a connection-oriented server to indicate that it is willing to receive connections.

```
int listen(int sockfd, int backlog);
```

It is usually executed after both the `socket` and `bind` system calls, and immediately before the `accept` system call. The *backlog* argument specifies how many connection requests can be queued by the system while it waits for the server to execute the `accept` system call. This argument is usually specified as 5, the maximum value currently allowed.

Recall our description of a concurrent connection-oriented server in Chapter 4. In the time that it takes a server to handle the request of an `accept` (the time required for the server to `fork` a child process and then have the parent process execute another `accept`), it is possible for additional connection requests to arrive from other clients. What the *backlog* argument refers to is this queue of pending requests for connections.

`accept` System Call

After a connection-oriented server executes the `listen` system call described above, an actual connection from some client process is waited for by having the server execute the `accept` system call.

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

int accept(int sockfd, struct sockaddr *peer, int *addrlen);
```

`accept` takes the first connection request on the queue and creates another socket with the same properties as *sockfd*. If there are no connection requests pending, this call blocks the caller until one arrives. (We discuss how a socket can be specified as non-blocking in Section 6.11.)

The *peer* and *addrlen* arguments are used to return the address of the connected peer process (the client). *addrlen* is called a value–result argument: the caller sets its value before the system call, and the system call stores a result in the variable. Often these value–result arguments are integers that the caller sets to the size of a buffer, with the system call changing this value on the return to the actual amount of data stored in the buffer. For this system call the caller sets *addrlen* to the size of the `sockaddr` structure whose address is passed as the *peer* argument. On return, *addrlen* contains the actual number of bytes that the system call stores in the *peer* argument. For the Internet and XNS address structures, since their sizes are constant (16 bytes), the value that we store

in the *addrlen* argument (16) is the actual number of bytes that the system call returns in the *peer* argument. The size of the structure can differ from the actual size of the address returned with Unix domain addresses.

This system call returns up to three values: an integer return code that is either an error indication or a new socket descriptor, the address of the client process (*peer*), and the size of this address (*addrlen*).

`accept` automatically creates a new socket descriptor, assuming the server is a concurrent server. If this is the case, the typical scenario is

```
int   sockfd, newsockfd;

if ( (sockfd = socket( ... )) < 0)
        err_sys("socket error");
if (bind(sockfd, ... ) < 0)
        err_sys("bind error");
if (listen(sockfd, 5) < 0)
        err_sys("listen error");

for ( ; ; ) {
        newsockfd = accept(sockfd, ... );   /* blocks */
        if (newsockfd < 0)
                err_sys("accept error");

        if (fork() == 0) {
                close(sockfd);          /* child */
                doit(newsockfd);        /* process the request */
                exit(0);
        }

        close(newsockfd);               /* parent */
}
```

When a connection request is received and accepted, the process `forks`, with the child process servicing the connection and the parent process waiting for another connection request. The new socket descriptor returned by `accept` refers to a complete association

{protocol, local-addr, local-process, foreign-addr, foreign-process}

All five elements of the 5-tuple associated with `newsockfd` have been filled in on return from `accept`. On the other hand, the `sockfd` argument that is passed to `accept` only has three elements of the 5-tuple filled in. The *foreign-addr* and *foreign-process* are still unspecified, and remain so after `accept` returns. This allows the original process (the parent) to `accept` another connection using `sockfd`, without having to create another socket descriptor. Since most connection-oriented servers are concurrent servers, and not iterative servers, the system goes ahead and creates a new socket automatically as part of the `accept` system call. If we wanted an iterative server, the scenario would be

```
int   sockfd, newsockfd;

if ( (sockfd = socket( ... ) < 0)
        err_sys("socket error");
bind(sockfd, ... ) < 0)
        err_sys("bind error");
listen(sockfd, 5) < 0)
        err_sys("listen error");

for ( ; ; ) {
        newsockfd = accept(sockfd, ... );   /* blocks */
        if (newsockfd < 0)
                err_sys("accept error");

        doit(newsockfd);      /* process the request */
        close(newsockfd);
}
```

Here the server handles the request using the connected socket descriptor, `newsockfd`. It then terminates the connection with a `close` and waits for another connection using the original descriptor, `sockfd`, which still has its *foreign-addr* and *foreign-process* unspecified.

`send`, `sendto`, `recv` and `recvfrom` System Calls

These system calls are similar to the standard `read` and `write` system calls, but additional arguments are required.

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

int send(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char *buff, int nbytes, int flags,
           struct sockaddr *to, int addrlen);

int recv(int sockfd, char *buff, int nbytes, int flags);

int recvfrom(int sockfd, char *buff, int nbytes, int flags,
             struct sockaddr *from, int *addrlen);
```

The first three arguments, *sockfd*, *buff*, and *nbytes*, to the four system calls are similar to the first three arguments for `read` and `write`.

The *flags* argument is either zero, or is formed by or'ing one of the following constants:

MSG_OOB	send or receive out-of-band data
MSG_PEEK	peek at incoming message (recv or recvfrom)
MSG_DONTROUTE	bypass routing (send or sendto)

The MSG_PEEK flag lets the caller look at the data that's available to be read, without having the system discard the data after the recv or recvfrom returns. We'll discuss the MSG_OOB option in Section 6.14 and the MSG_DONTROUTE option in Section 6.11. The *to* argument for sendto specifies the protocol-specific address of where the data is to be sent. Since this address is protocol-specific, its length must be specified by *addrlen*. The recvfrom system call fills in the protocol-specific address of who sent the data into *from*. The length of this address is also returned to the caller in *addrlen*. Note that the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value–result argument).

All four system calls return the length of the data that was written or read as the value of the function. In the typical use of recvfrom, with a connectionless protocol, the return value is the length of the datagram that was received.

close System Call

The normal Unix close system call is also used to close a socket.

```
int close(int fd);
```

If the socket being closed is associated with a protocol that promises reliable delivery (e.g., TCP or SPP), the system must assure that any data within the kernel that still has to be transmitted or acknowledged, is sent. Normally the system returns from the close immediately, but the kernel still tries to send any data already queued.

Later in this chapter we'll describe the SO_LINGER socket option. This socket option allows a process to specify that either: (a) the close should try to send any queued data, or (b) any data queued to be sent should be flushed.

Byte Ordering Routines

The following four functions handle the potential byte order differences between different computer architectures and different network protocols. We detailed some of the differences in Chapter 4.

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

u_long htonl(u_long hostlong);

u_short htons(u_short hostshort);

u_long ntohl(u_long netlong);

u_short ntohs(u_short netshort);
```

These functions were designed for the Internet protocols. Fortunately, the XNS protocols use the same byte ordering as the Internet. On those systems that have the same byte ordering as the Internet protocols (Motorola 68000-based systems, for example), these four functions are null macros. The conversions done by these functions are shown in Figure 6.9.

htonl	convert host–to–network, long integer
htons	convert host–to–network, short integer
ntohl	convert network–to–host, long integer
ntohs	convert network–to–host, short integer

Figure 6.9 Byte ordering functions.

These four functions all operate on unsigned integer values, although they work just as well on signed integers. Implicit in these functions is that a short integer occupies 16 bits and a long integer 32 bits.

Byte Operations

There are multibyte fields in the various socket address structures that need to be manipulated. Some of these fields, however, are not C integer fields, so some other technique must be used to operate on them portably.

4.3BSD defines the following three routines that operate on user-defined byte strings. By user-defined we mean they are not the standard C character strings (which are always terminated by a null byte). The user-defined byte strings can have null bytes within them and these do *not* signify the end of the string. Instead, we must specify the length of each string as an argument to the function.

```
bcopy(char *src, char *dest, int nbytes);

bzero(char *dest, int nbytes);

int bcmp(char *ptr1, char *ptr2, int nbytes);
```

`bcopy` moves the specified number of bytes from the source to the destination. Note that the order of the two pointer arguments is different from the order used by the standard I/O `strcpy` function. The `bzero` function writes the specified number of null

bytes to the specified destination. `bcmp` compare two arbitrary byte strings. The return value is zero if the two user-defined byte strings are identical, otherwise its nonzero. Note that this differs from the return value from the standard I/O `strcmp` function.

We'll use these three functions throughout the remaining chapters, usually to operate on socket address structures.

System V has three similar functions, `memcpy`, `memset`, and `memcmp`. But most socket libraries for System V supply functions with the 4.3BSD names, for compatibility. Details of the System V functions are on the *memory*(3) manual page.

Address Conversion Routines

As mentioned in Section 5.2, an Internet address is usually written in the dotted-decimal format, for example `192.43.235.1`. The following functions convert between the dotted-decimal format and an `in_addr` structure.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *ptr);

char *inet_ntoa(struct in_addr inaddr);
```

The first of these, `inet_addr` converts a character string in dotted-decimal notation to a 32-bit Internet address.† The `inet_ntoa` function does the reverse conversion.

Similar functions exist for handling XNS addresses, but there is not as clear a standard for representing XNS addresses in a readable form. The format of an XNS address is typically three fields, delimited by a separator character:

<network-ID><separator><host-ID><separator><port#>

For example, we write

```
123:02.07.01.00.a1.62:6001
```

using a colon as the field separator and a period as the byte separator for the host ID field. The function `ns_addr` defined below is lenient in what it accepts. First it tries to divide the character string into one to three fields, accepting either a period, colon, or pound sign as the separator. Each field is then scanned for byte separators, either a colon or a period. (Obviously you can't use the same character as the field separator and the byte separator.)

† Note that this function returns an unsigned long integer, when it should return an `in_addr` structure. Some systems that are based on 4.3BSD have corrected this.

```
#include <sys/types.h>
#include <netns/ns.h>

struct ns_addr ns_addr(char *ptr);

char *ns_ntoa(struct ns_addr ns);
```

The first of these converts a character string representation of an XNS network ID, host ID, and port number into an `ns_addr` structure. The second function does the reverse conversion. This function generates strings using a period to separate the three fields, with each field in hexadecimal.

6.6 A Simple Example

Now we'll use the elementary system calls from the previous section to provide 12 complete programs, showing examples of client–servers using each of the three protocol families available with 4.3BSD. We have 12 programs since we have a connection-oriented server, a connection-oriented client, a connectionless server and a connectionless client for each of the three protocol families (Internet, XNS, and Unix domain). The programs do the following:

1. The client reads a line from its standard input and writes the line to the server.
2. The server reads a line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.

This is an example of what is called an echo server. While we develop our own implementation of an echo server, most TCP/IP implementations provide such a server, using both TCP and UDP. See RFC 862 [Postel 1983a] for the official specification. Similarly, most XNS implementations also provide an echo server that is different from our implementation. We'll present a client for the standard XNS echo server in Section 11.3.

A pair of client–server programs to echo input lines is a good example of a network application. All the steps normally required to implement any client–server are illustrated by this example. All you need to do with this echo example, to expand it into your own application, is change what the server does with the input it receives from its clients.

In all these examples, we have "hard-coded" protocol-specific constants such as addresses and ports. There are two reasons for this. First, you should understand exactly what needs to be stored in the protocol-specific address structures. Second, we have not yet covered the library functions provided by 4.3BSD that make this more portable. These functions are covered in Section 8.2.

In these examples we have coded the connection-oriented Internet and Unix servers as concurrent servers, and the connection-oriented XNS server as an iterative server. This is to show examples of both types of servers.

Utility Routines

Stream sockets exhibit a behavior with the `read` and `write` system calls that differs from normal file I/O. A `read` or `write` on a socket might input or output fewer bytes than requested, but this is not an error condition. The reason is that buffer limits might be reached for the socket in the kernel and all that is required is for the caller to invoke the `read` or `write` system call again, to input or output the remaining bytes. Some versions of Unix also exhibit this behavior when writing more than 4096 bytes to a pipe.

We show three functions below that we'll use whenever we read or write to or from a stream socket. A datagram socket does not have this problem, since each I/O operation corresponds to a single packet.

```
/*
 * Read "n" bytes from a descriptor.
 * Use in place of read() when fd is a stream socket.
 */

int
readn(fd, ptr, nbytes)
register int    fd;
register char   *ptr;
register int    nbytes;
{
        int     nleft, nread;

        nleft = nbytes;
        while (nleft > 0) {
                nread = read(fd, ptr, nleft);
                if (nread < 0)
                        return(nread);          /* error, return < 0 */
                else if (nread == 0)
                        break;                  /* EOF */

                nleft -= nread;
                ptr   += nread;
        }
        return(nbytes - nleft);         /* return >= 0 */
}
```

The following function writes to a stream socket:

```
/*
 * Write "n" bytes to a descriptor.
 * Use in place of write() when fd is a stream socket.
 */

int
writen(fd, ptr, nbytes)
register int    fd;
register char   *ptr;
register int    nbytes;
{
```

```
        int     nleft, nwritten;

        nleft = nbytes;
        while (nleft > 0) {
                nwritten = write(fd, ptr, nleft);
                if (nwritten <= 0)
                        return(nwritten);               /* error */

                nleft -= nwritten;
                ptr   += nwritten;
        }
        return(nbytes - nleft);
}
```

We use the following function to read a line from a stream socket. In our examples we'll be exchanging Unix text lines between the client and server.

```
/*
 * Read a line from a descriptor.  Read the line one byte at a time,
 * looking for the newline.  We store the newline in the buffer,
 * then follow it with a null (the same as fgets(3)).
 * We return the number of characters up to, but not including,
 * the null (the same as strlen(3)).
 */

int
readline(fd, ptr, maxlen)
register int    fd;
register char   *ptr;
register int    maxlen;
{
        int     n, rc;
        char    c;

        for (n = 1; n < maxlen; n++) {
                if ( (rc = read(fd, &c, 1)) == 1) {
                        *ptr++ = c;
                        if (c == '\n')
                                break;
                } else if (rc == 0) {
                        if (n == 1)
                                return(0);      /* EOF, no data read */
                        else
                                break;          /* EOF, some data was read */
                } else
                        return(-1);     /* error */
        }

        *ptr = 0;
        return(n);
}
```

Note that our `readline` function issues one `read` system call for every byte of data. This is inefficient. Section 7.1 of Kernighan and Pike [1984], for example, shows

that this requires about ten times more CPU time than issuing a system call for every 10 bytes of data. What we would like to do is buffer the data by issuing a `read` system call to read as much data as we can, and then examine the buffer one byte at a time. Indeed, this is what the standard I/O library does. But to avoid having to worry about the possible interactions of the standard I/O library with a socket, we'll use the system call directly. It is possible to use the standard I/O library with a socket, but we have to be cognizant about the buffering that is taking place, and any interactions that can have with our application.

The three functions shown above are used throughout the text. We'll now present four functions that are common to the clients and servers shown in this section.

The following function is used by the three connection-oriented servers. It is an echo function that reads a line from the stream socket and writes it back to the socket.

```
/*
 * Read a stream socket one line at a time, and write each line back
 * to the sender.
 *
 * Return when the connection is terminated.
 */

#define MAXLINE 512

str_echo(sockfd)
int     sockfd;
{
        int     n;
        char    line[MAXLINE];

        for ( ; ; ) {
                n = readline(sockfd, line, MAXLINE);
                if (n == 0)
                        return;         /* connection terminated */
                else if (n < 0)
                        err_dump("str_echo: readline error");

                if (writen(sockfd, line, n) != n)
                        err_dump("str_echo: writen error");
        }
}
```

The following function is used by the three connection-oriented clients:

```
/*
 * Read the contents of the FILE *fp, write each line to the
 * stream socket (to the server process), then read a line back from
 * the socket and write it to the standard output.
 *
 * Return to caller when an EOF is encountered on the input file.
 */

#include        <stdio.h>
#define MAXLINE 512
```

```
str_cli(fp, sockfd)
register FILE   *fp;
register int    sockfd;
{
        int     n;
        char    sendline[MAXLINE], recvline[MAXLINE + 1];

        while (fgets(sendline, MAXLINE, fp) != NULL) {
                n = strlen(sendline);
                if (writen(sockfd, sendline, n) != n)
                        err_sys("str_cli: writen error on socket");

                /*
                 * Now read a line from the socket and write it to
                 * our standard output.
                 */

                n = readline(sockfd, recvline, MAXLINE);
                if (n < 0)
                        err_dump("str_cli: readline error");

                fputs(recvline, stdout);
        }

        if (ferror(fp))
                err_sys("str_cli: error reading file");
}
```

The following function is used by the three connectionless servers. By passing the address of the actual socket address structure to this function, it works with all three protocol families. Since the size of the structure can differ between protocol families, we also pass its size to this function, as it is needed for the `recvfrom` system call.

```
/*
 * Read a datagram from a connectionless socket and write it back to
 * the sender.
 *
 * We never return, as we never know when a datagram client is done.
 */

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

#define MAXMESG 2048

dg_echo(sockfd, pcli_addr, maxclilen)
int             sockfd;
struct sockaddr *pcli_addr;     /* ptr to appropriate sockaddr_XX structure */
int             maxclilen;      /* sizeof(*pcli_addr) */
{
        int     n, clilen;
        char    mesg[MAXMESG];

        for ( ; ; ) {
```

```
                clilen = maxclilen;
                n = recvfrom(sockfd, mesg, MAXMESG, 0, pcli_addr, &clilen);
                if (n < 0)
                        err_dump("dg_echo: recvfrom error");

                if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n)
                        err_dump("dg_echo: sendto error");
        }
}
```

The following function is for the connectionless clients. It is similar to the one for a connection-oriented client, with the `writen` calls replaced by `sendto` and the `readn` calls replaced by `recvfrom`. Also, we need the address of the actual socket address structure and its size for the datagram system calls.

```
/*
 * Read the contents of the FILE *fp, write each line to the
 * datagram socket, then read a line back from the datagram
 * socket and write it to the standard output.
 *
 * Return to caller when an EOF is encountered on the input file.
 */

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

#define MAXLINE 512

dg_cli(fp, sockfd, pserv_addr, servlen)
FILE            *fp;
int             sockfd;
struct sockaddr *pserv_addr;    /* ptr to appropriate sockaddr_XX structure */
int             servlen;        /* actual sizeof(*pserv_addr) */
{
        int     n;
        char    sendline[MAXLINE], recvline[MAXLINE + 1];

        while (fgets(sendline, MAXLINE, fp) != NULL) {
                n = strlen(sendline);
                if (sendto(sockfd, sendline, n, 0, pserv_addr, servlen) != n)
                        err_dump("dg_cli: sendto error on socket");

                /*
                 * Now read a message from the socket and write it to
                 * our standard output.
                 */

*/
                n = recvfrom(sockfd, recvline, MAXLINE, 0,
                                (struct sockaddr *) 0, (int *) 0);
                if (n < 0)
                        err_dump("dg_cli: recvfrom error");
                recvline[n] = 0;        /* null terminate */
                fputs(recvline, stdout);
```

```
        }

        if (ferror(fp))
                err_dump("dg_cli: error reading file");
}
```

TCP Example

Our TCP example follows the flow of system calls that we diagramed in Figure 6.2 for a connection-oriented client and server. First we show the header file that we use for both the TCP and UDP examples, `inet.h`.

```
/*
 * Definitions for TCP and UDP client/server programs.
 */

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

#define SERV_UDP_PORT   6543
#define SERV_TCP_PORT   6543
#define SERV_HOST_ADDR  "192.43.235.6"  /* host addr for server */

char    *pname;
```

Our choice of the TCP port number is arbitrary. It must be greater than 1023, should be greater than 5000, and must not conflict with any other TCP server's port. (We describe the 4.3BSD port number conventions in more detail in Section 6.8.) The `SERV_HOST_ADDR` constant corresponds to the host being used by the author. As mentioned earlier, there are better ways of handling these magic numbers and constants, but for now we want to concentrate on the socket system calls themselves.

The server program is as follows:

```
/*
 * Example of server using TCP protocol.
 */

#include        "inet.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd, newsockfd, clilen, childpid;
        struct sockaddr_in      cli_addr, serv_addr;

        pname = argv[0];

        /*
```

```
        * Open a TCP socket (an Internet stream socket).
        */

       if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
               err_dump("server: can't open stream socket");

       /*
        * Bind our local address so that the client can send to us.
        */

       bzero((char *) &serv_addr, sizeof(serv_addr));
       serv_addr.sin_family      = AF_INET;
       serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
       serv_addr.sin_port        = htons(SERV_TCP_PORT);

       if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
               err_dump("server: can't bind local address");

       listen(sockfd, 5);

       for ( ; ; ) {
               /*
                * Wait for a connection from a client process.
                * This is an example of a concurrent server.
                */

               clilen = sizeof(cli_addr);
               newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                                                                &clilen);
               if (newsockfd < 0)
                       err_dump("server: accept error");

               if ( (childpid = fork()) < 0)
                       err_dump("server: fork error");

               else if (childpid == 0) {       /* child process */
                       close(sockfd);          /* close original socket */
                       str_echo(newsockfd);    /* process the request */
                       exit(0);
               }

               close(newsockfd);               /* parent process */
       }
}
```

We specify the Internet address for the server's `bind` as the constant `INADDR_ANY`. This tells the system that we will accept a connection on any Internet interface on the system, if the system is multihomed.

The client program is as follows:

```
/*
 * Example of client using TCP protocol.
 */
```

```
#include        "inet.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd;
        struct sockaddr_in      serv_addr;

        pname = argv[0];

        /*
         * Fill in the structure "serv_addr" with the address of the
         * server that we want to connect with.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family      = AF_INET;
        serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
        serv_addr.sin_port        = htons(SERV_TCP_PORT);

        /*
         * Open a TCP socket (an Internet stream socket).
         */

        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                err_sys("client: can't open stream socket");

        /*
         * Connect to the server.
         */

        if (connect(sockfd, (struct sockaddr *) &serv_addr,
                                                    sizeof(serv_addr)) < 0
                err_sys("client: can't connect to server");

        str_cli(stdin, sockfd);         /* do it all */

        close(sockfd);
        exit(0);
}
```

UDP Example

Our UDP client and server programs follow the system call flow that we diagramed in Figure 6.3 for a connectionless protocol.

The client calls `bind` after creating its socket, to have the system assign any local address to the client's socket. In some instances this specific step isn't required, if the system automatically assigns a local address when the `sendto` system call is used for

the first time. But some connectionless protocols, such as the Unix domain datagram protocol, don't do this (for reasons which we discuss later in this section). Therefore, as both a good programming habit, and to provide consistency between protocols, we'll always call `bind` for connectionless clients.

This UDP example is not reliable. If a datagram disappears, neither the client or server are aware of it. To provide reliability for UDP requires additional user-level code that we'll develop in Section 8.4.

First the server program.

```
/*
 * Example of server using UDP protocol.
 */

#include        "inet.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd;
        struct sockaddr_in      serv_addr, cli_addr;

        pname = argv[0];

        /*
         * Open a UDP socket (an Internet datagram socket).
         */

        if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
                err_dump("server: can't open datagram socket");

        /*
         * Bind our local address so that the client can send to us.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family      = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        serv_addr.sin_port        = htons(SERV_UDP_PORT);

        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
                err_dump("server: can't bind local address");

        dg_echo(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));

                /* NOTREACHED */
}
```

As with the TCP server example, the Internet address for the `bind` is specified as `INADDR_ANY`.

The client program:

```
/*
 * Example of client using UDP protocol.
 */

#include        "inet.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd;
        struct sockaddr_in      cli_addr, serv_addr;

        pname = argv[0];

        /*
         * Fill in the structure "serv_addr" with the address of the
         * server that we want to send to.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family      = AF_INET;
        serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
        serv_addr.sin_port        = htons(SERV_UDP_PORT);

        /*
         * Open a UDP socket (an Internet datagram socket).
         */

        if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
                err_dump("client: can't open datagram socket");

        /*
         * Bind any local address for us.
         */

        bzero((char *) &cli_addr, sizeof(cli_addr));    /* zero out */
        cli_addr.sin_family      = AF_INET;
        cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        cli_addr.sin_port        = htons(0);
        if (bind(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0)
                err_dump("client: can't bind local address");

        dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr

        close(sockfd);
        exit(0);
}
```

In assigning a local address for the client using `bind`, we set the Internet address to `INADDR_ANY` and the 16-bit Internet port to zero. Setting the port to zero causes the system to assign the client some unused port in the range 1024 through 5000. The system

guarantees that the assigned port is unique on the local system, so that the datagrams returned by the server to this port are delivered to the correct client. When we send a datagram to the server, the system determines which Internet interface to use to contact the server, and sets the client's 32-bit Internet address accordingly.

SPP Example

The XNS examples are similar to the Internet examples, the only exception being the handling of the XNS address structures.

First we show the header file used by both the SPP and IDP programs, `xns.h`.

```
/*
 * Definitions for SPP and IDP client/server programs.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netns/ns.h>

#define SPP_SERV_ADDR   "123:02.07.01.00.a1.62:6001"
#define IDP_SERV_ADDR   "123:02.07.01.00.a1.62:6000"
                                /* <netid>:<hostid>:<port> */
#define SERV_SPP_PORT   6001
#define SERV_IDP_PORT   6000

char            *pname;
struct ns_addr  ns_addr();      /* BSD library routine */
```

Our choice of port numbers is arbitrary. Note that the SPP and IDP port numbers are different, while our Internet example used the same port number for the TCP port and the UDP port. Recall from Section 5.2 that TCP and UDP ports are independent on a system (they are demultiplexed at the transport layer). XNS port numbers, however, are not independent as they are demultiplexed at the network layer.

The `SPP_SERV_ADDR` and `IDP_SERV_ADDR` values correspond to the XNS addresses of the systems used for this example.

As mentioned at the beginning of this section, we have coded this server as an iterative server. It completely handles one client before accepting another connection. You should compare this with the TCP example shown earlier in this section, which we coded as a concurrent server.

```
/*
 * Example of server using SPP protocol.
 */

#include        "xns.h"

main(argc, argv)
int     argc;
char    *argv[];
```

```
{
        int                     sockfd, newsockfd, clilen;
        struct sockaddr_ns      cli_addr, serv_addr;

        pname = argv[0];

        /*
         * Open a SPP socket (an XNS stream socket).
         */

        if ( (sockfd = socket(AF_NS, SOCK_STREAM, 0)) < 0)
                err_dump("server: can't open stream socket");

        /*
         * Bind our local address so that the client can send to us.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sns_family      = AF_NS;
        serv_addr.sns_addr.x_port = htons(SERV_SPP_PORT);

        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < (
                err_dump("server: can't bind local address");

        listen(sockfd, 5);

        for ( ; ; ) {
                /*
                 * Wait for a connection from a client process,
                 * then process it without fork()'ing.
                 * This is an example of an iterative server.
                 */

                clilen = sizeof(cli_addr);
                newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                                                                &clilen);
                if (newsockfd < 0)
                        err_dump("server: accept error");

                str_echo(newsockfd);    /* returns when connection is closed */

                close(newsockfd);
        }
}
```

Before binding its well-known address, the server zeroes its address structure and then stores its family and port, leaving the network ID and host ID as zero. This is similar to setting the 32-bit Internet network ID and host ID to `INADDR_ANY`. It tells the system that the server is willing to accept a connection from any host.

The client program:

```
/*
 * Example of client using SPP protocol.
 */

#include        "xns.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd;
        struct sockaddr_ns      serv_addr;

        pname = argv[0];

        /*
         * Fill in the structure "serv_addr" with the XNS address of the
         * server that we want to connect with.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sns_family = AF_NS;
        serv_addr.sns_addr   = ns_addr(SPP_SERV_ADDR);
                               /* stores net-ID, host-ID and port */

        /*
         * Open a SPP socket (an XNS stream socket).
         */

        if ( (sockfd = socket(AF_NS, SOCK_STREAM, 0)) < 0)
                err_sys("client: can't open stream socket");

        /*
         * Connect to the server.
         */

        if (connect(sockfd, (struct sockaddr *) &serv_addr,
                                                sizeof(serv_addr)) < 0)
                err_sys("client: can't connect to server");

        str_cli(stdin, sockfd);         /* do it all */

        close(sockfd);
        exit(0);
}
```

IDP Example

As with our UDP example, the IDP programs that follow are not reliable. The IDP server program:

```
/*
 * Example of server using XNS IDP protocol.
 */

#include        "xns.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd;
        struct sockaddr_ns      serv_addr, cli_addr;

        pname = argv[0];

        /*
         * Open an IDP socket (an XNS datagram socket).
         */

        if ( (sockfd = socket(AF_NS, SOCK_DGRAM, 0)) < 0)
                err_dump("server: can't open datagram socket");

        /*
         * Bind our local address so that the client can send to us.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sns_family       = AF_NS;
        serv_addr.sns_addr.x_port  = htons(SERV_IDP_PORT);

        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
                err_dump("server: can't bind local address");

        dg_echo(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));
                /* NOTREACHED */
}
```

The IDP client program:

```
/*
 * Example of client using XNS IDP protocol.
 */

#include        "xns.h"

main(argc, argv)
int     argc;
char    *argv[];
{
```

```
        int                     sockfd;
        struct sockaddr_ns      cli_addr, serv_addr;

        pname = argv[0];

        /*
         * Fill in the structure "serv_addr" with the address of the
         * server that we want to send to.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sns_family = AF_NS;
        serv_addr.sns_addr   = ns_addr(IDP_SERV_ADDR);
                               /* stores net-ID, host-ID and port */

        /*
         * Open an IDP socket (an XNS datagram socket).
         */

        if ( (sockfd = socket(AF_NS, SOCK_DGRAM, 0)) < 0)
                err_dump("client: can't open datagram socket");

        /*
         * Bind any local address for us.
         */

        bzero((char *) &cli_addr, sizeof(cli_addr));    /* zero out */
        cli_addr.sns_family = AF_NS;
        if (bind(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0)
                err_dump("client: can't bind local address");

        dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

        close(sockfd);
        exit(0);
}
```

Unix Stream Example

The Unix stream protocol example is similar to both the TCP and SPP examples. Here is the header file that we use for both the stream and datagram examples, `unix.h`.

```
/*
 * Definitions for UNIX domain stream and datagram client/server programs.
 */

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

#define UNIXSTR_PATH    "./s.unixstr"
#define UNIXDG_PATH     "./s.unixdg"
```

```
#define UNIXDG_TMP      "/tmp/dg.XXXXXX"

char    *pname;
```

The server program:

```
/*
 * Example of server using UNIX domain stream protocol.
 */

#include        "unix.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd, newsockfd, clilen, childpid, servlen;
        struct sockaddr_un      cli_addr, serv_addr;

        pname = argv[0];

        /*
         * Open a socket (a UNIX domain stream socket).
         */

        if ( (sockfd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
                err_dump("server: can't open stream socket");

        /*
         * Bind our local address so that the client can send to us.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sun_family = AF_UNIX;
        strcpy(serv_addr.sun_path, UNIXSTR_PATH);
        servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);

        if (bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
                err_dump("server: can't bind local address");

        listen(sockfd, 5);

        for ( ; ; ) {
                /*
                 * Wait for a connection from a client process.
                 * This is an example of a concurrent server.
                 */

                clilen = sizeof(cli_addr);
                newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                                                                &clilen);
                if (newsockfd < 0)
                        err_dump("server: accept error");
```

```
                if ( (childpid = fork()) < 0)
                        err_dump("server: fork error");

                else if (childpid == 0) {       /* child process */
                        close(sockfd);          /* close original socket */
                        str_echo(newsockfd);    /* process the request */
                        exit(0);
                }

                close(newsockfd);               /* parent process */
        }
}
```

With a Unix domain address structure, we have to calculate the length of the structure for the `bind` system call, since it depends on the length of the pathname specified. We specify the length as the size of the `sun_family` variable plus the number of characters in the pathname.

The client program:

```
/*
 * Example of client using UNIX domain stream protocol.
 */

#include        "unix.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd, servlen;
        struct sockaddr_un      serv_addr;

        pname = argv[0];

        /*
         * Fill in the structure "serv_addr" with the address of the
         * server that we want to send to.
         */

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sun_family = AF_UNIX;
        strcpy(serv_addr.sun_path, UNIXSTR_PATH);
        servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);

        /*
         * Open a socket (an UNIX domain stream socket).
         */

        if ( (sockfd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
                err_sys("client: can't open stream socket");

        /*
         * Connect to the server.
         */
```

```
        if (connect(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
                err_sys("client: can't connect to server");

        str_cli(stdin, sockfd);         /* do it all */

        close(sockfd);
        exit(0);
}
```

Unix Datagram Example

The Unix datagram protocol example presents an interesting problem that we haven't encountered yet. Since Unix domain sockets are identified by pathnames of actual files in the host's filesystem, the client has to form a unique pathname to use when it `binds` its local address. When using UDP or IDP, the local port is specified by a 16-bit integer that only has significance within the kernel. Therefore we let the system assign our local port, leaving the responsibility to the system to guarantee its uniqueness on the local host. But in the Unix domain we're dealing with pathnames that get turned into actual files by the `bind` system call, so it is our responsibility to pick a local address and guarantee its uniqueness on the local host. While this function could have been delegated to the kernel (as is done with UDP and IDP), since side effects could occur (such as creating a file in a directory in which the caller doesn't have permission to delete in), the designers chose to leave this to the application.

In this example we use the `mktemp` function in the standard C library to create a name for the client's socket, based on the client's process ID. Note that we must try to create a unique name for the client's socket, since there is a possibility that multiple clients are using the same server at the same time. With unique client addresses, each server response is returned to the correct client process.†

The server program:

```
/*
 * Example of server using UNIX domain datagram protocol.
 */

#include        "unix.h"

main(argc, argv)
int     argc;
```

† There is a feature in the 4.3BSD implementation of Unix domain datagrams that limits the number of characters of this filename that are transmitted along with the datagram, to 14 characters. If the client `binds` a local address with more than 14 characters in the filename, that file, of type socket, is correctly created in the filesystem, but only the first 14 characters of its name are sent to the server. This means the server won't be able to respond to the client. Furthermore, if the filename is less than 14 characters, extraneous characters are usually appended to the name, which also prevents the server from responding to the client. We have purposely chosen the pathname `/tmp/dg.XXXXXX` to be exactly 14 characters.

```
char    *argv[];
{
        int                     sockfd, servlen;
        struct sockaddr_un      serv_addr, cli_addr;

        pname = argv[0];

        /*
         * Open a socket (a UNIX domain datagram socket).
         */

        if ( (sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
                err_dump("server: can't open datagram socket");

        /*
         * Bind our local address so that the client can send to us.
         */

        unlink(UNIXDG_PATH);    /* in case it was left from last time */
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sun_family = AF_UNIX;
        strcpy(serv_addr.sun_path, UNIXDG_PATH);
        servlen = sizeof(serv_addr.sun_family) + strlen(serv_addr.sun_path);

        if (bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
                err_dump("server: can't bind local address");

        dg_echo(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));
                /* NOTREACHED */
}
```

The client program:

```
/*
 * Example of client using UNIX domain datagram protocol.
 */

#include        "unix.h"

main(argc, argv)
int     argc;
char    *argv[];
{
        int                     sockfd, clilen, servlen;
        char                    *mktemp();
        struct sockaddr_un      cli_addr, serv_addr;

        pname = argv[0];

        /*
         * Fill in the structure "serv_addr" with the address of the
         * server that we want to send to.
         */
```

```
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sun_family = AF_UNIX;
        strcpy(serv_addr.sun_path, UNIXDG_PATH);
        servlen = sizeof(serv_addr.sun_family) + strlen(serv_addr.sun_path);

        /*
         * Open a socket (a UNIX domain datagram socket).
         */

        if ( (sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
                err_dump("client: can't open datagram socket");

        /*
         * Bind a local address for us.
         * In the UNIX domain we have to choose our own name (that
         * should be unique).  We'll use mktemp() to create a unique
         * pathname, based on our process id.
         */

        bzero((char *) &cli_addr, sizeof(cli_addr));    /* zero out */
        cli_addr.sun_family = AF_UNIX;
        strcpy(cli_addr.sun_path, UNIXDG_TMP);
        mktemp(cli_addr.sun_path);
        clilen = sizeof(cli_addr.sun_family) + strlen(cli_addr.sun_path);

        if (bind(sockfd, (struct sockaddr *) &cli_addr, clilen) < 0)
                err_dump("client: can't bind local address");

        dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, servlen);

        close(sockfd);
        unlink(cli_addr.sun_path);
        exit(0);
}
```

6.7 Advanced Socket System Calls

We now proceed to the more advanced socket system calls, which we need in later sections of this chapter, and in some of the later chapters of this text.

`readv` and `writev` System Calls

4.3BSD provides what are called *scatter read* and *gather write* variants of the standard `read` and `write` system calls that we described in Section 2.3. What these two variants provide is the ability to read into or write from noncontiguous buffers.

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

int writev(int fd, struct iovec iov[], int iovcount);

int readv(int fd, struct iovec iov[], int iovcount);
```

These two system calls use the following structure that is defined in `<sys/uio.h>`:

```
struct iovec {
  caddr_t  iov_base;     /* starting address of buffer */
  int      iov_len;      /* size of buffer in bytes */
};
```

The `writev` system call writes the buffers specified by *iov[0]*, *iov[1]*, through *iov[iovcount–1]*. The `readv` system call does the input equivalent. `readv` always fills one buffer (as specified by the `iov_len` value) before proceeding to the next buffer in the *iov* array. Both system calls return the total number of bytes read or written.

Consider, as an example, a hypothetical function named `write_hdr` that writes a buffer preceded by some header. The actual format of the header doesn't concern us.

```
int
write_hdr(fd, buff, nbytes)
int     fd;
char    *buff;
int     nbytes;
{
        int                n;
        struct hdr_info    header;

        /* ... set up the header as required ... */

        if (write(fd, header, sizeof(header)) != sizeof(header))
                return(-1);

        if (write(fd, buff, nbytes) != nbytes)
                return(-1);

        return(nbytes);
}
```

Here two `write` system calls are used. Another way to do this is to allocate a temporary buffer, build the header in the beginning of the buffer, then copy the data into the buffer and do a single `write` of the entire buffer. A third way to do it (to avoid the extra `write` or to avoid the copy operation) is to force the caller to allocate space at the beginning of their buffer for the header. None of these three alternatives is optimal, which is why the `writev` system call (and its `readv` counterpart) were introduced.

The types of operation described here (adding information to either the beginning or end of a user's buffer) is just a form of encapsulation. The `writev` alternative for our example is

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

int
write_hdr(fd, buff, nbytes)
int     fd;
char    *buff;
int     nbytes;
{
        struct hdr_info   header;
        struct iovec      iov[2];

        /* ... set up the header as required ... */

        iov[0].iov_base = (char *) &header;
        iov[0].iov_len  = sizeof(header);

        iov[1].iov_base = buff;
        iov[1].iov_len  = nbytes;

        if (writev(fd, &iov[0], 2) != sizeof(header) + nbytes)
                return(-1);

        return(nbytes);
}
```

The `readv` and `writev` system calls can be used with any descriptor, not just sockets, although they are not portable beyond 4.3BSD. (One could write user functions named `readv` and `writev` that emulate these 4.3BSD system calls.)

The `writev` system call is an atomic operation. This is important if the descriptor refers to a record-based entity, such as a datagram socket or a magnetic tape drive. For example, a `writev` to a datagram socket produces a single datagram, whereas multiple `writes` would produce multiple datagrams. The only correct way to emulate `writev` is to copy all the data into a single buffer and then execute a single `write` system call.

`sendmsg` and `recvmsg` System Calls

These two system calls are the most general of all the read and write system calls.

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

int sendmsg(int sockfd, struct msghdr msg[], int flags);

int recvmsg(int sockfd, struct msghdr msg[], int flags);
```

These use the msghdr structure that is defined in <sys/socket.h>

```
struct msghdr {
  caddr_t       msg_name;          /* optional address */
  int           msg_namelen;       /* size of address */
  struct iovec  *msg_iov;          /* scatter/gather array */
  int           msg_iovlen;        /* # elements in msg_iov */
  caddr_t       msg_accrights;     /* access rights sent/recvd */
  int           msg_accrightslen;
};
```

The msg_name and msg_namelen fields are used when the socket is not connected, similar to the final two arguments to the recvfrom and sendto system calls. The msg_name field can be specified as a NULL pointer if a name is either not required or not desired. The msg_iov and msg_iovlen fields are used for scatter read and gather write operations, as described above for the readv and writev system calls. The final two elements of the structure, msg_accrights and msg_accrightslen deal with the passing and receiving of access rights between processes. We discuss this in Section 6.10 when we explain the passing of file descriptors between processes. The flags argument is the same as with the send and recv system calls, which we described in Section 6.5.

Figure 6.10 compares the five different read and write function groups.

System call	Any descriptor	Only socket descriptor	Single read/write buffer	Scatter/ gather read/write	Optional flags	Optional peer address	Optional access rights
read, write	•		•				
readv, writev	•			•			
recv, send		•	•		•		
recvfrom, sendto		•	•		•	•	
recvmsg, sendmsg		•		•	•	•	•

Figure 6.10 Read and write system call variants.

getpeername System Call

This system call returns the name of the peer process that is connected to a given socket.

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

int getpeername(int sockfd, struct sockaddr *peer, int *addrlen);
```

When we say that this system call returns the "name" of the peer process, we mean that it returns the *foreign-addr* and *foreign-process* elements of the 5-tuple associated with *sockfd*. Note that the final argument is a value–result argument.

The original 4.3BSD release did not support this system call for Unix domain sockets, but fixes exist to correct this limitation.

`getsockname` System Call

This system call returns the name associated with a socket.

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

int getsockname(int sockfd, struct sockaddr *peer, int *addrlen);
```

This call returns the *local-addr* and *local-process* elements of an association. As with the `getpeername` system call, the final argument is a value–result argument.

The original 4.3BSD release did not support this system call for Unix domain sockets, but fixes exist to correct this limitation.

`getsockopt` and `setsockopt` System Calls

These two system calls manipulate the options associated with a socket. We'll describe them in more detail in Section 6.11, along with two other system calls, `fcntl` and `ioctl`, that can modify the properties of a socket.

`shutdown` System Call

The normal way to terminate a network connection is to call the `close` system call. As we mentioned earlier, `close` normally attempts to deliver any data that is still to be sent. But the `shutdown` system call provides more control over a full-duplex connection.

```
int shutdown(int sockfd, int howto);
```

If the *howto* argument is 0, no more data can be received on the socket. A value of 1 causes no more output to be allowed on the socket. A value of 2 causes both sends and receives to be disallowed.

Remember that a socket is usually a *full-duplex* communication path. The data flowing in one direction is logically independent of the data going in the other direction. This is why `shutdown` allows either direction to be closed, independent of the other direction.

`select` System Call

This system call can be used when dealing with multiple descriptors. We'll discuss it in detail in Section 6.13.

6.8 Reserved Ports

There are two ways for a process to have an Internet port or an XNS port assigned to a socket.

- The process can request a specific port. This is typical for servers that need to assign a well-known port to a socket. All our server examples shown in Section 6.6 do this.
- The process can let the system automatically assign a port. For both the Internet domain and the XNS domain, specifying a port number of zero before calling `bind` requests the system to do this. Both our UDP and IDP client examples do this.

4.3BSD supports the concept of *reserved ports* in both the Internet and XNS domains. In the Internet domain, any TCP or UDP port in the range 1–1023 is reserved, and in the XNS domain ports in the range 1–2999 are reserved. A process is not allowed to `bind` a reserved port unless its effective user ID is zero (the superuser).

4.3BSD provides a library function that assigns a reserved TCP stream socket to its caller:

```
int rresvport(int *aport);
```

This function creates an Internet stream socket and `binds` a reserved port to the socket. The socket descriptor is returned as the value of the function, unless an error occurs, in which case −1 is returned. Note that the argument to this function is the address of an integer (a value–result argument), not an integer value. The integer pointed to by *aport* is the first port number that the function attempts to `bind`. The caller typically initializes the starting port number to `IPPORT_RESERVED-1`. (The value of the constant `IPPORT_RESERVED` is defined to be 1024 in `<netinet/in.h>`.) If this `bind` fails with an `errno` of `EADDRINUSE`, then this function decrements the port number and tries again. If it finally reaches port 512 and finds it already in use, it sets `errno` to `EAGAIN`, and returns −1. If this function returns successfully, it not only returns the socket as the value of the function, but the port number is also returned in the location pointed to by *aport*.

Figure 6.11 summarizes the assignment of ports in the Internet and XNS domains.

	Internet	XNS
reserved ports	1–1023	1–2999
ports automatically assigned by system	1024–5000	3000–65535
ports assigned by `rresvport()`	512–1023	

Figure 6.11 Port assignment in the Internet and XNS domains.

Note that all ports assigned by standard Internet applications (FTP, TELNET, TFTP, SMTP, etc.) are between 1 and 255 and are therefore reserved. The servers for these applications must have superuser privileges when they create their socket and `bind` their well-known address. Also, the ports between 256 and 511 are considered reserved by 4.3BSD, but they are not currently used by any standard Internet application, and they are never allocated by the `rresvport` function.

The system doesn't automatically assign an Internet port greater than 5000. It leaves these ports for user-developed, nonprivileged servers. This provides a higher degree of certainty that a server can assign itself its well-known port, since any of the ports between 1024 and 5000 might be in use by some client.

Finally, note that any process that wants to obtain a privileged UDP, IDP, or SPP port has to do the same steps that `rresvport` does. There does not exist a standard library function similar to `rresvport` for these three protocols.

The concept of *reserved ports* as described above only handles the `binding` of ports to unbound sockets by the system. It is up to the application program (the server) that receives a request from a client with a reserved port, to consider the request as special or not. The server can obtain the address of the client using the `getpeername` system call. A connection-oriented server can also obtain the client's address from the *peer* argument of the `accept` system call, and a datagram server can obtain the address of the client from the *from* argument of the `recvfrom` system call. In Section 9.2 we consider reserved ports and their use by typical 4.3BSD servers for authentication.

6.9 Stream Pipes

We introduced the term *stream pipe* in Section 6.5 when describing the `socketpair` system call. We consider a stream pipe to be similar to a pipe, but a stream pipe is *bidirectional.* Furthermore, stream pipes are used in both 4.3BSD and in System V Release 3 for passing file descriptors between processes. Therefore we'll provide a function named `s_pipe` to create a stream pipe and have it look just like the `pipe` system call.

We also have the option of associating a name with a stream pipe, so we can provide two versions: *unnamed* stream pipes and *named* stream pipes. Figure 6.12 compares the four types of pipes that we've described.

	Unidirectional	Bidirectional	4.3BSD	System V
pipe	•		yes	yes
named pipe (FIFO)	•		some	yes
stream pipe		•	yes	yes
named stream pipe		•	yes	yes

Figure 6.12 Comparison of pipes, FIFOs, stream pipes, and named stream pipes.

Pipes and named pipes were described in Chapter 3.

A function to create an unnamed stream pipe is trivial for 4.3BSD.

```
/*
 * Create an unnamed stream pipe.
 */

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

int                     /* returns 0 if all OK, -1 if error (with errno set) */
s_pipe(fd)
int     fd[2];          /* two file descriptors returned through here */
{
        return( socketpair(AF_UNIX, SOCK_STREAM, 0, fd) );
}
```

Our function to create a named stream pipe first creates an unnamed stream pipe, then associates a name with one end of it, using the `bind` system call. This function requires another argument: the name within the filesystem to be associated with the stream pipe.

```
/*
 * Create a named stream pipe.
 */

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

int                     /* returns 0 if all OK, -1 if error (with errno set) */
ns_pipe(name, fd)
char    *name;          /* user-specified name to assign to the stream pipe */
int     fd[2];          /* two file descriptors returned through here */
{
        int                     len;
        struct sockaddr_un      unix_addr;

        if (s_pipe(fd) < 0)     /* first create an unnamed stream pipe */
                return(-1);

        unlink(name);   /* remove the name, if it already exists */

        bzero((char *) &unix_addr, sizeof(unix_addr));
        unix_addr.sun_family = AF_UNIX;
        strcpy(unix_addr.sun_path, name);
        len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);

        if (bind(fd[0], (struct sockaddr *) &unix_addr, len) < 0)
                return(-1);

        return(0);
}
```

As we mentioned in Section 6.3, the filename that is specified for a socket that is created in the Unix domain (as we are doing to implement named stream pipes) refers to a socket, not a true Unix file. The only way another process can access this endpoint is through the socket system calls. We cannot, for example, use the `open` system call on these named stream pipes.

With stream pipes, both unnamed and named, the two descriptors returned by the functions `s_pipe` and `ns_pipe` are indistinguishable: both can be used for reading and writing, since these stream pipes are bidirectional. Recall with the `pipe` system call that only one descriptor is available for reading and the other is only available for writing.

6.10 Passing File Descriptors

Recall from Chapter 2 that for most Unix systems the only way to pass an open file from one process to another is for a parent process to open a file and then either `fork` or `exec` another process. There is normally no way for one process to pass an open file to some other, arbitrary (i.e., unrelated) process, or for a child to pass an open file to its parent process. 4.3BSD and System V Release 3 both provide a mechanism to do this. When we say "open file" we mean "open descriptor," since either a file descriptor or a socket descriptor can be passed. Since devices are handled using the same descriptors as files, descriptors that refer to devices can also be passed. When we use the term "file descriptor" in this section, realize that 4.3BSD descriptors are not restricted to actual files.

Recall our picture of the open file table in the kernel for a given process (Figure 2.12). It is a vector that is indexed by the file descriptor to obtain a pointer into the kernel's file table. When we say we want to pass an open file from one process to another, what we want to do is take an open file from the sending process and generate another pointer to the file table entry for that file. Pictorially, we want the data structures as shown in Figure 6.13.

Notice that the actual descriptor value is generally different between the sending process and the receiving process. Figure 6.13 shows, for example, that the descriptor could be 3 in the sending process, but the descriptor in the receiving process could be 4, if that is the first available value. Nevertheless, entry *fd[3]* in the sending process and entry *fd[4]* in the receiving process both point to the same file table entry in the kernel, allowing the two processes to share the same file. Typically the sending process closes the file after it has been sent to the receiving process.

To pass a descriptor from one process to another, the two processes must be connected with a Unix domain socket. This connection can be either a stream socket or a datagram socket. Given the comments we presented earlier about the potential buffer problems with a datagram protocol, we'll use a stream protocol in our examples. We'll use the `s_pipe` function from the previous section to create a stream pipe which is really a Unix domain stream socket. The only way to pass a descriptor is for the sending process to use the `sendmsg` system call and then the receiving process must use the

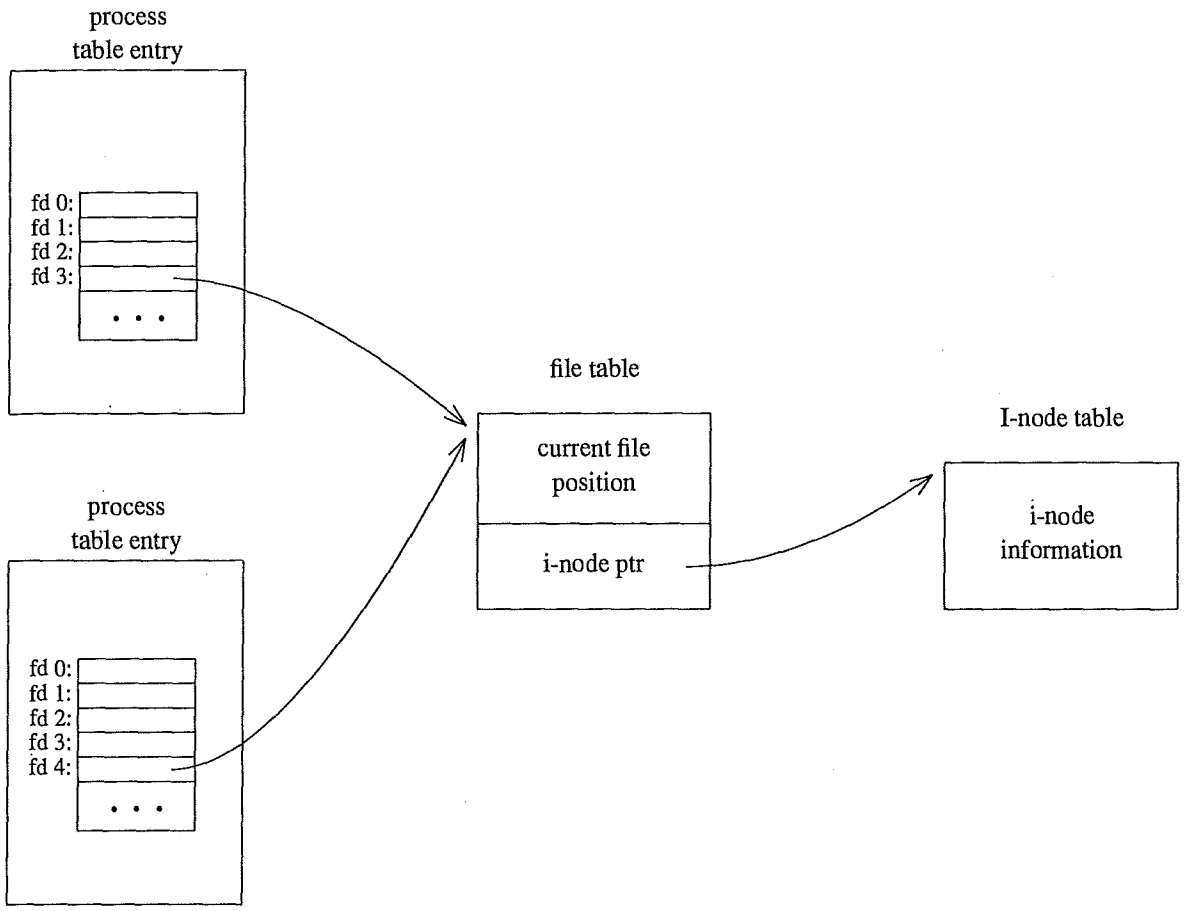


Figure 6.13 Sharing a file table entry between two processes.

`recvmsg` system call. These two system calls are the only ones that support the concept of "access rights," which is how descriptors are passed.

The term "access rights" indicates that a process has the right to access a system-maintained object. In this case the object is a file, device, or socket that the system has granted access to by opening the object and returning a file descriptor to the process. The process that has this access right (the sending process that obtained the right from the kernel) passes it to another process. Although we can imagine other access rights that might be passed from one process to another (access to a portion of memory, for example), 4.3BSD only supports the passing of file descriptors.

There are two instances where the ability to pass a file descriptor can be useful. (These ideas come from Presotto and Ritchie [1985].)

1. A process `forks` a child process and the child then `execs` another process to open a particular file. The `execed` process does whatever it has to do to open the file and then passes the open file descriptor back to the original process. Since the `execed` process is a child process of the original process, there is no way for it to pass an open file back to the parent, other than passing an open file as we describe in this section.

2. A connection server can be invoked to handle all network connections. Instead of using `fork` and `exec` as described above, this scenario requires any client process to connect with a well-known server. The client writes its request to the server using this connection (a Unix domain stream socket, for example) and the server responds with an open file descriptor or an error indication.

An example of scenario 1 is the establishment of a network connection using an automatic dialer. In 4.3BSD, for example, the programs `tip` (also known as `cu`) and `uucico` (the program that does most of the work for UUCP) both include functions to handle all possible dialers. The `uucico` program also includes code to establish TCP connections with the other system. Whenever a new dialer is introduced, a new function has to be written to handle it and all programs wanting to access the new dialer have to be recompiled and link edited. Another solution is to have a network access program that is `exec`ed by both `tip` and `uucico` to handle all the details involved with establishing a network connection. This new program passes the open file descriptor (or an error indication) back to the invoking process. Any arguments required by the network access program (dialer type, telephone number, network address, etc.) can be passed as command line arguments in the `exec` system call. Note that this solution does involve a `fork` and `exec`, while the current technique only involves a function call.

An Example

The following example shows how to pass and receive file descriptors in a 4.3BSD environment. First we have a `main` function that takes zero or more command line arguments that specify files to be copied to the standard output. If the program is called `mycat` then it can be invoked as

```
mycat /etc/passwd
```

to print the password file on your terminal.

```
/*
 * Catenate one or more files to standard output.
 * If no files are specified, default to standard input.
 */

#define BUFFSIZE        4096

main(argc, argv)
int     argc;
char    *argv[];
{
        int             fd, i, n;
        char            buff[BUFFSIZE];
        extern char     *pname;

        pname = argv[0];
        argv++; argc--;
```

```
        fd = 0;         /* default to stdin if no arguments */
        i = 0;
        do {
                if (argc > 0 && (fd = my_open(argv[i], 0)) < 0) {
                        err_ret("can't open %s", argv[i]);
                        continue;
                }

                while ( (n = read(fd, buff, BUFFSIZE)) > 0)
                        if (write(1, buff, n) != n)
                                err_sys("write error");
                if (n < 0)
                        err_sys("read error");

        } while (++i < argc);
        exit(0);
}
```

If the call to the function `my_open` shown above is replaced with a call to the standard library function named `open`, the program works fine. It is similar to the standard Unix `cat` command. What we'll do, however, is provide our own function named `my_open`. Our version goes through the steps listed in scenario 1 above: it `fork`s a copy of itself and the child process `exec`s another program to do the actual `open` system call, returning a file descriptor to the parent process. Here is our version of the `open` function.

```
/*
 * Open a file, returning a file descriptor.
 *
 * This function is similar to the UNIX open() system call,
 * however here we invoke another program to do the actual
 * open(), to illustrate the passing of open files
 * between processes.
 */

int
my_open(filename, mode)
char    *filename;
int     mode;
{
        int             fd, childpid, sfd[2], status;
        char            argsfd[10], argmode[10];
        extern int      errno;

        if (s_pipe(sfd) < 0)            /* create an unnamed stream pipe */
                return(-1);             /* errno will be set */

        if ( (childpid = fork()) < 0)
                err_sys("can't fork");

        else if (childpid == 0) {       /* child process */
                close(sfd[0]);          /* close the end we don't use */
                sprintf(argsfd, "%d", sfd[1]);
                sprintf(argmode, "%d", mode);
                if (execl("./openfile", "openfile", argsfd, filename,
```

```
                        argmode, (char *) 0) < 0)
                            err_sys("can't execl");
        }

        /* parent process - wait for the child's execl() to complete */

        close(sfd[1]);                  /* close the end we don't use */
        if (wait(&status) != childpid)
                err_dump("wait error");
        if ((status & 255) != 0)
                err_dump("child did not exit");
        status = (status >> 8) & 255;           /* child's exit() argument */
        if (status == 0) {
                fd = recvfile(sfd[0]);          /* all OK, receive fd */
        } else {
                errno = status; /* error, set errno value from child's errno */
                fd = -1;
        }

        close(sfd[0]);          /* close the stream pipe */
        return(fd);
}
```

The program that is execed by the child is the one that does the open system call. Its pathname was specified as `./openfile` in the function shown above.

```
/*
 *      openfile  <socket-descriptor-number>  <filename>  <mode>
 *
 * Open the specified file using the specified mode.
 * Return the open file descriptor on the specified socket descriptor
 * (which the invoker has to set up before exec'ing us).
 * We exit() with a value of 0 if all is OK, otherwise we pass back the
 * errno value as our exit status.
 */

main(argc, argv)
int     argc;
char    *argv[];
{
        int             fd;
        extern int      errno;
        extern char     *pname;

        pname = argv[0];

        if (argc != 4)
                err_quit("openfile <sockfd#> <filename> <mode>");

        /*
         * Open the file.
         */

        if ( (fd = open(argv[2], atoi(argv[3]))) < 0)
                exit( (errno > 0) ? errno : 255 );
```

```
        /*
         * And pass the descriptor back to caller.
         */

        exit( sendfile(atoi(argv[1]), fd) );
}
```

Both of the programs above use the following two functions to send and receive the file descriptor being passed. We have coded the actual calls to `sendmsg` and `recvmsg` as separate functions, to allow their inclusion into other programs that might want to pass descriptors.

```
/*
 * Pass a file descriptor to another process.
 * Return 0 if OK, otherwise return the errno value from the
 * sendmsg() system call.
 */

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

int
sendfile(sockfd, fd)
int     sockfd;         /* UNIX domain socket to pass descriptor on */
int     fd;             /* the actual fd value to pass */
{
        struct iovec    iov[1];
        struct msghdr   msg;
        extern int      errno;

        iov[0].iov_base = (char *) 0;           /* no data to send */
        iov[0].iov_len  = 0;
        msg.msg_iov             = iov;
        msg.msg_iovlen          = 1;
        msg.msg_name            = (caddr_t) 0;
        msg.msg_accrights       = (caddr_t) &fd;    /* address of descriptor */
        msg.msg_accrightslen = sizeof(fd);      /* pass 1 descriptor */

        if (sendmsg(sockfd, &msg, 0) < 0)
                return( (errno > 0) ? errno : 255 );

        return(0);
}

/*
 * Receive a file descriptor from another process.
 * Return the file descriptor if OK, otherwise return -1.
 */

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

```
int
recvfile(sockfd)
int     sockfd;         /* UNIX domain socket to receive descriptor on */
{
        int             fd;
        struct iovec    iov[1];
        struct msghdr   msg;

        iov[0].iov_base = (char *) 0;           /* no data to receive */
        iov[0].iov_len  = 0;
        msg.msg_iov          = iov;
        msg.msg_iovlen       = 1;
        msg.msg_name         = (caddr_t) 0;
        msg.msg_accrights    = (caddr_t) &fd;   /* address of descriptor */
        msg.msg_accrightslen = sizeof(fd);      /* receive 1 descriptor */

        if (recvmsg(sockfd, &msg, 0) < 0)
                return(-1);

        return(fd);
}
```

6.11 Socket Options

There are a multitude of ways to set options that affect a socket.

- the `setsockopt` system call
- the `fcntl` system call
- the `ioctl` system call

We mentioned the `fcntl` and `ioctl` system calls in Section 2.3, and describe their effect on sockets later in this section.

`getsockopt` and `setsockopt` System Calls

These two system calls apply only to sockets.

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

int getsockopt(int sockfd, int level, int optname, char *optval, int *opt

int setsockopt(int sockfd, int level, int optname, char *optval, int optle
```

The *sockfd* argument must refer to an open socket descriptor. The *level* specifies who in the system is to interpret the option: the general socket code, the TCP/IP code, or the XNS code. Some options are protocol-specific and others are applicable to all sockets.

The *optval* argument is a pointer to a user variable from which an option value is set by `setsockopt`, or into which an option value is returned by `getsockopt`. Although this argument is type coerced into a '`char *`', none of the supported options are character values. The "data type" column in Figure 6.14 shows the data type of what the *optval* pointer must point to for each option. The *optlen* argument to `setsockopt` specifies the size of the variable. The *optlen* argument to `getsockopt` is a value–result parameter that we set to the size of the *optval* variable before the call. This size is then set by the system on return to specify the amount of data stored into the *optval* variable. This ability to handle variable-length options is only used by a single option currently: `IP_OPTIONS`. All the other options described below pass a fixed amount of data between the kernel and the user process.

Figure 6.14 provides a summary of all options that can be queried by `getsockopt` or set by `setsockopt`. There are two types of options: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values). The column labeled "flag" specifies if the option is a flag option. When calling `getsockopt` for these flag options, *optval* is an integer. The value returned in *optval* is zero if the option is disabled, or nonzero if the option is enabled. Similarly, `setsockopt` requires a nonzero *optval* to turn the option on, and a zero value to turn the option off. If the "flag" column does not contain a "•" then the option is used to pass a value of the specified data type between the user process and the system.

level	*optname*	get	set	Description	flag	Data type
IPPROTO_IP	IP_OPTIONS	•	•	options in IP header		
IPPROTO_TCP	TCP_MAXSEG	•		get TCP maximum segment size		int
	TCP_NODELAY	•	•	don't delay send to coalesce packets	•	int
NSPROTO_SPP	SO_HEADERS_ON_INPUT	•	•	pass SPP header on input	•	int
	SO_HEADERS_ON_OUTPUT	•	•	pass SPP header on output	•	int
	SO_DEFAULT_HEADERS	•	•	set default SPP header for output		struct sphdr
	SO_LAST_HEADER	•		fetch last SPP header		struct sphdr
	SO_MTU	•	•	SPP MTU value		u_short
NSPROTO_PE	SO_SEQNO	•		generate unique PEX ID		long
NSPROTO_RAW	SO_HEADERS_ON_INPUT	•	•	pass IDP header on input	•	int
	SO_HEADERS_ON_OUTPUT	•	•	pass IDP header on output	•	int
	SO_DEFAULT_HEADERS	•	•	set default IDP header for output		struct idp
	SO_ALL_PACKETS	•	•	pass all IDP packets to user	•	int
	SO_NSIP_ROUTE		•	specify IP route for XNS		struct nsip_req
SOL_SOCKET	SO_BROADCAST	•	•	permit sending of broadcast msgs	•	int
	SO_DEBUG	•	•	turn on debugging info recording	•	int
	SO_DONTROUTE	•	•	just use interface addresses	•	int
	SO_ERROR	•		get error status and clear		int
	SO_KEEPALIVE	•	•	keep connections alive	•	int
	SO_LINGER	•	•	linger on close if data present		struct linger
	SO_OOBINLINE	•	•	leave received OOB data in-line	•	int
	SO_RCVBUF	•	•	receive buffer size		int
	SO_SNDBUF	•	•	send buffer size		int
	SO_RCVLOWAT	•	•	receive low-water mark		int
	SO_SNDLOWAT	•	•	send low-water mark		int
	SO_RCVTIMEO	•	•	receive timeout		int
	SO_SNDTIMEO	•	•	send timeout		int
	SO_REUSEADDR	•	•	allow local address reuse	•	int
	SO_TYPE	•		get socket type		int
	SO_USELOOPBACK	•	•	bypass hardware when possible	•	int

Figure 6.14 Socket options for `getsockopt` and `setsockopt`.

Let's first show a simple program that both fetches an option and sets an option.

```
/*
 * Example of getsockopt() and setsockopt().
 */

#include        <sys/types.h>
#include        <sys/socket.h>          /* for SOL_SOCKET and SO_xx values */
#include        <netinet/in.h>          /* for IPPROTO_TCP value */
#include        <netinet/tcp.h>         /* for TCP_MAXSEG value */

main()
{
        int     sockfd, maxseg, sendbuff, optlen;

        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
```

```
                err_sys("can't create socket");

        /*
         * Fetch and print the TCP maximum segment size.
         */

        optlen = sizeof(maxseg);
        if (getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, (char *) &maxseg,
                       &optlen) < 0)
                err_sys("TCP_MAXSEG getsockopt error");
        printf("TCP maxseg = %d\n", maxseg);

        /*
         * Set the send buffer size, then fetch it and print its value.
         */

        sendbuff = 16384;       /* just some number for example purposes */
        if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &sendbuff,
                                                    sizeof(sendbuff)) < 0)
                err_sys("SO_SNDBUF setsockopt error");

        optlen = sizeof(sendbuff);
        if (getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &sendbuff,
                                                    &optlen) < 0)
                err_sys("SO_SNDBUF getsockopt error");
        printf("send buffer size = %d\n", sendbuff);
}
```

If we compile and execute this program, its output is

```
TCP maxseg = 512
send buffer size = 16384
```

The most confusing thing about the `getsockopt` and `setsockopt` system calls is the handling of the final argument.

The following list gives additional details on the options that affect a socket. When we say *allows us to specify* we are referring to the `setsockopt` system call. We say *when requested* to refer to the `getsockopt` system call. When a given option can only be used with a specific type of socket, the socket type (such as `SOCK_DGRAM`) is listed in parentheses first. For example, the `TCP_MAXSEG` option only applies to an Internet `SOCK_STREAM` socket.

`IP_OPTIONS` (Internet `SOCK_STREAM` or `SOCK_DGRAM`.) Allows us to set specific options in the IP header. Requires intimate knowledge of the IP header. See Section 7.8 of Comer [1988] for a description of the possible IP options. We'll use this option in our remote command server in Section 14.3 and our remote login server in Section 15.12. Both of these servers check for client connections that specify IP options, and if encountered, record the fact in the system log and disable the options.

TCP_MAXSEG (Internet SOCK_STREAM.) Returns the maximum segment size in use for the socket. The typical value for a 4.3BSD socket using an Ethernet is 1024 bytes. Note that the value for this option can only be examined, it cannot be set by us, since the system decides what size to use.

Note in the example program shown above that the value printed for this option was 512 and not 1024. This is because the socket was not yet connected, so TCP does not know the type of interface being used. It defaults the size to 512 until the socket is connected when it decides on the size to use.

TCP_NODELAY (Internet SOCK_STREAM.) When TCP is being used for a remote login (which we discuss in Chapter 15) there will be many small data packets sent from the client's system to the server. Each packet can contain a single character that the client enters, which is sent to the server for echoing and processing. On a fast LAN these small packets are not a problem, since the capacity of the network (its bandwidth) is usually adequate to handle all the packets. On a slower WAN, however, it is desirable to reduce the number of these small packets, to reduce the traffic on the network. The scheme used by 4.3BSD is to allow only a single small packet to be outstanding on a given TCP connection at any time. (For additional details on the 4.3BSD TCP output algorithms, see Section 12.7 of Leffler et al. [1989].) On a fast LAN this isn't a problem since the round-trip time for a 1-character packet is usually less than the time between a user entering successive characters on a terminal. On a slower WAN, what happens is the client's TCP buffers input characters until the previous small packet is acknowledged. One problem, however, is if the client is sending small packets that are not keystrokes being entered on a terminal to be echoed by the server. This can happen if the client is sending input from a mouse, for example, on a windowed terminal. The client's input can consist of small packets that are not echoed by the server. The TCP_NODELAY option is used for these clients, to defeat the buffering algorithm described above, to allow the client's TCP to send small packets as soon as possible.

SO_HEADERS_ON_INPUT

(XNS SOCK_DGRAM.) When set, the first 30 bytes of the data returned by the read and receive system calls is the IDP header.

(XNS SOCK_STREAM or SOCK_SEQPACKET.) When set, the first 12 bytes of the data returned by the read and receive system calls is the SPP header.

SO_HEADERS_ON_OUTPUT

(XNS SOCK_DGRAM.) When set, the first 30 bytes of the data passed to the send and write system calls must be the IDP header.

(XNS SOCK_STREAM or SOCK_SEQPACKET.) When set, the first 12 bytes of the data passed to the send and write system calls must be the SPP header.

SO_DEFAULT_HEADERS

This option provides a way for a user process to set and examine the XNS bridge fields that we described in Section 5.3.

(XNS SOCK_DGRAM.) Allows us to specify a default IDP header for outgoing packets. The actual variable pointed to by the *optval* argument must be a `struct idp` (the structure defining an IDP header). The only value used by the system from this header is the packet type field. When requested, the system returns its default IDP header, with the following fields filled in: packet type, local address, and foreign address. We'll show an example using this option in Section 11.3.

(XNS SOCK_STREAM or SOCK_SEQPACKET.) Allows us to specify a default SPP header for outgoing packets. The actual variable pointed to by the *optval* argument must be a `struct sphdr` (the structure defining an SPP header). The only values used by the system from this header are the End-of-Message bit and the Datastream Type. When requested, the system returns its default SPP header.

SO_LAST_HEADER

(XNS SOCK_STREAM or SOCK_SEQPACKET.) When requested, returns the most recent SPP header received. The actual variable pointed to by the *optval* argument must be a `struct sphdr` (the structure defining an SPP header).

SO_MTU

(XNS SOCK_STREAM or SOCK_SEQPACKET.) By default the SPP MTU is 534 bytes, the Xerox standard. This allows a maximum IDP packet size of 576 bytes, allowing 30 bytes for the IDP header and 12 bytes for the SPP header. Note that any value other than this default is frowned on by an actual Xerox system. This option applies only to SPP sockets, since the BSD implementation of XNS allows IDP packets up to the maximum size of the underlying interface. For an Ethernet, this maximum IDP packet size is 1500 bytes. Again, however, these larger than normal IDP packets can create problems with true Xerox systems.

SO_SEQNO (XNS SOCK_DGRAM.) The XNS Packet Exchange protocol requires a unique 32-bit ID field, called the transaction ID, with every packet. This ID should be unique among different processes on the same system. This socket option allows the caller to obtain a unique value. The kernel allocates these IDs to any requesting process, incrementing the ID each time a process needs one. Furthermore, the kernel initializes its starting ID whenever the system is rebooted, from a random source (usually the time-of-day clock). This way individual processes can obtain unique IDs and, if the system is taken down and rebooted quickly (while a packet is possibly being retransmitted by some other host on the network), the IDs provided after rebooting are different from those used earlier. Note that the value for this option can only be examined, it cannot be set by us.

SO_ALL_PACKETS (XNS SOCK_DGRAM.) When set, prevents the system from processing Error Protocol packets and SPP packets. Instead, these packets are passed to us.

SO_NSIP_ROUTE (XNS SOCK_DGRAM.) NSIP is an option that can be enabled when the 4.3BSD kernel is configured. If enabled, it allows XNS packets to be sent to another 4.3BSD system, encapsulated in Internet IP datagrams. This allows us, for example, to share a single data link on a WAN between Internet applications and XNS applications. Using this feature any two cooperating systems that are connected on an Internet using the TCP/IP protocols can also exchange XNS packets. This socket option allows us to specify an Internet address to be associated with an XNS address. We won't discuss this option any further in this text.

SO_BROADCAST (Internet or XNS SOCK_DGRAM.) Enables or disables the ability of the process to send broadcast messages. Broadcasting is only provided for datagram sockets and only on networks that support the concept of a broadcast message (the kernel does not provide a simulation of broadcasting through software).

SO_DEBUG Enables or disables low-level debugging within the kernel. This option allows the kernel to maintain a history of the recent packets that have been received or sent. With adequate knowledge of the kernel, this history can either be examined (by looking at kernel memory with a debugger) or printed on the console.

SO_DONTROUTE Specifies that outgoing messages are to bypass the normal routing mechanisms of the underlying protocol. Instead, the message is directed to the appropriate network interface, as specified by the network portion of the destination address. The equivalent of this

option can also be applied to individual datagrams using the MSG_DONTROUTE flag with the send, sendto, or sendmsg system calls.

SO_ERROR Returns to the caller the current contents of the variable so_error, which is defined in <sys/socketvar.h>. This variable holds the standard Unix error numbers (the same values found in the Unix errno variable) for the socket. This error variable is then cleared by the kernel.

SO_KEEPALIVE (Internet or XNS SOCK_STREAM.) Enables periodic transmissions on a connected socket, when no other data is being exchanged. If the other end does not respond to these messages, the connection is considered broken and the so_error variable is set to ETIMEDOUT.

SO_LINGER (Internet SOCK_STREAM.) This option determines what to do when unsent messages exist for a socket when the process executes a close on the socket. By default, the close returns immediately and the system attempts to deliver any unsent data. But if the linger option is set for the socket, the action taken depends on the linger time specified by the user. This option requires the following structure to be passed between the user process and the kernel. It is defined in <sys/socket.h>.

```
struct linger {
  int   l_onoff;   /* zero=off, nonzero=on */
  int   l_linger;  /* linger time, in seconds */
};
```

If the linger time is specified as zero, any remaining data to be sent is discarded when the socket is closed. If the linger time is nonzero, the system attempts to deliver any unsent data. Currently the actual value of a nonzero linger time is ignored, despite the comment in the structure definition.

SO_OOBINLINE (Internet SOCK_STREAM.) When set, specifies that out-of-band data also be placed in the normal input queue (i.e., in-line). When the out-of-band data is placed in-line, the MSG_OOB flag to the receive system calls is not needed to read the out-of-band data. This option applies only to TCP sockets, since the XNS SPP protocol specifies that out-of-band data also be placed in the normal input queue by default. We discuss out-of-band data in more detail in Section 6.14.

SO_RCVBUF and SO_SNDBUF
Specifies the size of the receive queue buffer or the send queue buffer for the socket. These options are only needed when we would like more buffer space than is provided by default. These options are not required for these cases, but they can improve

performance. For example, the BSD `rdump` command, which writes 32768-byte buffers using TCP, sets its `SO_SNDBUF` option value to 32768, overriding the default TCP send buffer of 4096 bytes. Similarly, the `/etc/rmt` daemon, which receives these buffers and writes them to tape, sets its `SO_RCVBUF` option to 32768 also. With 4.3BSD there is an upper limit of around 52,000 bytes for either of these buffer sizes. We'll see one use of these options when we look at the 4.3BSD remote tape server in Chapter 16.

`SO_RCVLOWAT` and `SO_SNDLOWAT`
: These two options specify the sizes of the receive and send low-water marks. These values are currently unused.

`SO_RCVTIMEO` and `SO_SNDTIMEO`
: These two options specify the receive and send timeout values. These values are not currently used for anything. Note that this does not imply that the reliable protocols do not use timers for detecting lost packets and lost connections. Timers are indeed used by TCP and SPP, for example, but these two socket options don't currently affect these timers.

`SO_REUSEADDR`
: (Internet `SOCK_STREAM` or `SOCK_DGRAM`.) Instructs the system to allow local addresses, the *local-process* portion of an association 5-tuple, to be reused. Normally the system does not allow a local address to be reused. When `connect` is called for the socket, the system still requires that the complete association be unique, as discussed earlier. We mentioned this requirement in Section 5.2 when discussing the use of TCP port numbers by FTP.

`SO_TYPE`
: Returns the socket type. The integer value returned is a value such as `SOCK_STREAM` or `SOCK_DGRAM`. This option is typically used by a process that inherits a socket when it is started.

`SO_USELOOPBACK`
: This option is unused.

`fcntl` System Call

This system call affects an open file, referenced by the *fd* argument.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, int arg);
```

As mentioned in Section 2.3, the *cmd* argument specifies the operation to be performed. Regarding sockets we are interested in the *cmd* values of `F_GETOWN`, `F_SETOWN`, `F_GETFL`, and `F_SETFL`.

The F_SETOWN command sets either the process ID or the process group ID to receive the SIGIO or SIGURG signals for the socket associated with *fd*. Every socket has an associated process group number, similar to the terminal process group number that we discussed in Chapter 2. For a socket, its process group number is initialized to zero and can be set by calling fcntl with the F_SETOWN command. (It can also be set by both the FIOSETOWN and SIOCSPGRP ioctls. As we said, there are many ways to set options that affect a socket.) The *arg* value for the F_SETOWN command can be either a positive integer, specifying a process ID, or a negative integer, specifying a process group ID. The F_GETOWN command returns, as the return value from the fcntl system call, either the process ID (a positive return value) or the process group ID (a negative value other than −1) associated with the socket. The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group (perhaps more than one) to receive the signal. This fcntl command is only available for terminals and sockets. We show an example of the F_SETOWN command in Section 6.12.

A file's flag bits are set and examined with the F_SETFL and F_GETFL commands. Figure 6.15 shows the *arg* values that can be used with the F_GETFL and F_SETFL commands.

arg	Meaning
FAPPEND	append on each write
FASYNC	signal process group when data ready
FCREAT	create if nonexistent
FEXCL	error if already created
FNDELAY	nonblocking I/O
FTRUNC	truncate to zero length

Figure 6.15 fcntl options for F_GETFL and F_SETFL commands.

Of these five values, only the following two are of interest to us now.

FNDELAY This option designates the socket as "nonblocking." An I/O request that cannot complete on a nonblocking socket is not done. Instead, return is made to the caller immediately and the global errno is set to EWOULDBLOCK.

This option affects the following system calls: accept, connect, read, readv, recv, recvfrom, recvmsg, send, sendto, sendmsg, write, and writev. Note that a connect on a connectionless socket cannot block, since all the system does is record the peer address that is to be used for future output. On a connection-oriented socket, however, the connect can take a while to execute, since it usually involves the exchange of actual information with the peer system. In this case a nonblocking socket returns immediately from the connect system call, but the errno value is set to EINPROGRESS instead of EWOULDBLOCK.

Note also that the output system calls (the five starting with `send` in the list above) do partial writes on a nonblocking socket when it makes sense (e.g., if the socket is a stream socket). When the socket has record boundaries associated with it (e.g., a datagram socket), if the entire record cannot be written, nothing is written.

`FASYNC` This option allows the receipt of asynchronous I/O signals. The `SIGIO` signal is sent to the process group of the socket when data is available to be read. We'll cover asynchronous I/O in the next section.

`ioctl` System Call

This system call affects an open file, referenced by the *fd* argument. The intent is for `ioctl`s to manipulate device options.

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, char *arg);
```

System V defines the second argument to be an `int` instead of an `unsigned long`, but that is not a problem, since the header file `<sys/ioctl.h>` defines the constants that should be used for this argument.

We can divide the *request*s into four categories.

- file operations
- socket operations
- routing operations
- interface operations

Figure 6.16 lists the *request*s provided by 4.3BSD, along with the data type of what the *arg* address must point to. (Notice that none of the requests use the character-pointer data type specified in the function prototype.) We now give a brief description of these options.

`FIOCLEX` Sets the close-on-`exec` flag for the file descriptor. Similar to the `F_SETFD` command of `fcntl` (with an *arg* of one).

`FIONCLEX` Clears the close-on-`exec` flag for the file descriptor. Similar to the `F_SETFD` command of `fcntl` (with an *arg* of zero).

`FIONBIO` Set or clear the nonblocking I/O flag for the file. This flag accomplishes the same affect as the `FNDELAY` argument for the `F_SETFL` command to the `fcntl` system call. Note one difference between the two system calls, however: with the `F_SETFL` command of `fcntl` we must specify all the flag values for the file each time we want to

Category	*request*	Description	Data type
file	FIOCLEX	set close-on-exec for fd	
	FIONCLEX	clear close-on-exec for fd	
	FIONBIO	set/clear nonblocking i/o	int
	FIOASYNC	set/clear asynchronous i/o	int
	FIONREAD	get # bytes to read	int
	FIOSETOWN	set owner	int
	FIOGETOWN	get owner	int
socket	SIOCSHIWAT	set high-water mark	int
	SIOCGHIWAT	get high-water mark	int
	SIOCSLOWAT	set low-water mark	int
	SIOCGLOWAT	get low-water mark	int
	SIOCATMARK	at out-of-band mark ?	int
	SIOCSPGRP	set process group	int
	SIOCGPGRP	get process group	int
routing	SIOCADDRT	add route	struct rtentry
	SIOCDELRT	delete route	struct rtentry
interface	SIOCSIFADDR	set ifnet address	struct ifreq
	SIOCGIFADDR	get ifnet address	struct ifreq
	SIOCSIFFLAGS	set ifnet flags	struct ifreq
	SIOCGIFFLAGS	get ifnet flags	struct ifreq
	SIOCGIFCONF	get ifnet list	struct ifconf
	SIOCSIFDSTADDR	set point-to-point address	struct ifreq
	SIOCGIFDSTADDR	get point-to-point address	struct ifreq
	SIOCGIFBRDADDR	get broadcast addr	struct ifreq
	SIOCSIFBRDADDR	set broadcast addr	struct ifreq
	SIOCGIFNETMASK	get net addr mask	struct ifreq
	SIOCSIFNETMASK	set net addr mask	struct ifreq
	SIOCGIFMETRIC	get IF metric	struct ifreq
	SIOCSIFMETRIC	set IF metric	struct ifreq
	SIOCSARP	set ARP entry	struct arpreq
	SIOCGARP	get ARP entry	struct arpreq
	SIOCDARP	delete ARP entry	struct arpreq

Figure 6.16 `ioctl` options for networking.

turn a specific flag on or off. With this `ioctl` call, however, we can turn a flag on or off, by specifying a zero or nonzero value for *arg*.

`FIOASYNC` Set or clear the flag that allows the receipt of asynchronous I/O signals (`SIGIO`). This flag accomplishes the same affect as the `FASYNC` argument for the `F_SETFL` command to the `fcntl` system call.

`FIONREAD` Returns in *arg* the number of bytes available to read from the file descriptor. This feature works for files, pipes, terminals and sockets.

`FIOSETOWN` Set either the process ID or the process group ID to receive the `SIGIO` and `SIGURG` signals. This *request* works only for terminals and sockets. This command is identical to an `fcntl` of `F_SETOWN`.

`FIOGETOWN` Get either the process ID or the process group ID that is set to receive the `SIGIO` and `SIGURG` signals. This *request* works only for terminals and sockets. This command is identical to an `fcntl` of `F_GETOWN`.

`SIOCSHIWAT` Set the high-water mark. This command is not currently implemented.

`SIOCGHIWAT` Get the high-water mark. This command is not currently implemented.

`SIOCSLOWAT` Set the low-water mark. This command is not currently implemented.

`SIOCGLOWAT` Get the low-water mark. This command is not currently implemented.

`SIOCATMARK` Return a zero or nonzero value, depending whether the specified socket's read pointer is currently at the out-of-band mark. We describe out-of-band data in more detail in Section 6.14.

`SIOCSPGRP` Equivalent to `FIOSETOWN` for a socket.

`SIOCGPGRP` Equivalent to `FIOGETOWN` for a socket.

The following *requests* are for lower level operations, usually relating to the actual network interface being used. We briefly mention their purpose. The interested reader should consult both of the references by Leffler et al. [1986a, 1986b] for additional information. Several of these commands are used by the network configuration program, `ifconfig`, which is described in Section 8 of the 4.3BSD Programmer's Manual. This program is usually invoked on system startup to initialize all the network interfaces on the system. Some of the commands related to routing are used by the BSD routing daemon, `routed`, which is also described in Section 8 of the BSD manual.

`SIOCADDRT` Add an entry to the interface routing table.

`SIOCDELRT` Delete an entry from the interface routing table.

`SIOCSIFADDR`
Set the interface address. Additionally the initialization function for the interface is also called.

`SIOCGIFADDR`
Get the interface address.

`SIOCSIFFLAGS`
Set the interface flags.

`SIOCGIFFLAGS`
Get the interface flags. The flags indicate, for example, if the interface is a point-to-point interface, if the interface supports broadcast addressing, if the interface is running, and so on.

`SIOCGIFCONF`
Get the interface configuration list. This command returns a list containing one `ifreq` structure for every interface currently is use by the

system. This command allows a user program to determine at run time, the interfaces on the system. A program can then go through this list and determine which interfaces it is interested in (all interfaces that are point-to-point Internet interfaces, for example).

`SIOCSIFDSTADDR`
: Set the point-to-point interface address.

`SIOCGIFDSTADDR`
: Get the point-to-point interface address.

`SIOCSIFBRDADDR`
: Set the broadcast address for the interface.

`SIOCGIFBRDADDR`
: Get the broadcast address for the interface.

`SIOCSIFNETMASK`
: Set the mask for the network portion of the interface address. For Internet addresses the network mask defines the netid portion of the 32-bit address. If this mask contains more bits that would normally be indicated by the class of Internet address (class A, B, or C) then subnets are being used. Refer to Section 5.2 for additional details on Internet addresses.

`SIOCGIFNETMASK`
: Get the interface network mask for an Internet address.

`SIOCGIFMETRIC`
: Get the interface routing metric. The routing metric is stored in the kernel for each interface, but is used by the routing protocol, the `routed` daemon.

`SIOCSIFMETRIC`
: Set the interface routing metric. The 4.3BSD kernel does not make policy decisions for routing. Instead, this is left to a user process, which then uses this `ioctl` to modify the kernel's routing tables. Refer to Section 11.5 of Leffler et al. [1989] for additional details.

`SIOCSARP` Set an entry in the Internet ARP (Address Resolution Protocol) table. The structure `arpreq` is defined in `<net/if_arp.h>`. Refer to the entry for `arp`(4P) in Section 4 of the BSD manual for additional information.

`SIOCGARP` Get an entry from the Internet ARP entry. The caller specifies an Internet address and this system call returns the corresponding Ethernet address.

`SIOCDARP` Delete an entry from the Internet ARP table. The caller specifies the Internet address for the entry to be deleted.

6.12 Asynchronous I/O

Asynchronous I/O allows the process to tell the kernel to notify it when a specified descriptor is ready for I/O. It is also called signal-driven I/O. The notification from the kernel to the user process takes place with a signal, the `SIGIO` signal.

To do asynchronous I/O, a process must perform the following three steps:

1. The process must establish a handler for the `SIGIO` signal. This is done by calling the `signal` system call, as described in Section 2.4.
2. The process must set the process ID or the process group ID to receive the `SIGIO` signals. This is done with the `fcntl` system call, with the `F_SETOWN` command, as described in Section 6.11.
3. The process must enable asynchronous I/O using the `fcntl` system call, with the `F_SETFL` command and the `FASYNC` argument.

To show an example of asynchronous I/O, let's first show a simple program that copies standard input to standard output.

```
/*
 * Copy standard input to standard output.
 */

#define BUFFSIZE        4096

main()
{
        int             n;
        char            buff[BUFFSIZE];

        while ( (n = read(0, buff, BUFFSIZE)) > 0)
                if (write(1, buff, n) != n)
                        err_sys("write error");

        if (n < 0)
                err_sys("read error");

        exit(0);
}
```

We now change this program to do the three steps listed, using asynchronous I/O.

```
/*
 * Copy standard input to standard output, using asynchronous I/O.
 */

#include        <signal.h>
#include        <fcntl.h>

#define BUFFSIZE        4096
```

```
int     sigflag;

main()
{
        int             n;
        char            buff[BUFFSIZE];
        int             sigio_func();

        signal(SIGIO, sigio_func);

        if (fcntl(0, F_SETOWN, getpid()) < 0)
                err_sys("F_SETOWN error");

        if (fcntl(0, F_SETFL, FASYNC) < 0)
                err_sys("F_SETFL FASYNC error");

        for ( ; ; ) {
                sigblock(sigmask(SIGIO));
                while (sigflag == 0)
                        sigpause(0);            /* wait for a signal */

                /*
                 * We're here if (sigflag != 0).  Also, we know that the
                 * SIGIO signal is currently blocked.
                 */

                if ( (n = read(0, buff, BUFFSIZE)) > 0) {
                        if (write(1, buff, n) != n)
                                err_sys("write error");
                } else if (n < 0)
                        err_sys("read error");
                else if (n == 0)
                        exit(0);                /* EOF */

                sigflag = 0;                    /* turn off our flag */

                sigsetmask(0);                  /* and reenable signals */
        }
}

int
sigio_func()
{
        sigflag = 1;            /* just set flag and return */

        /* the 4.3BSD signal facilities leave this handler enabled
           for any further SIGIO signals. */
}
```

This interrupt driven example works only for terminals (and would work for a socket too, if the appropriate code were added to create a socket) because of the restrictions of the 4.3BSD `F_SETOWN` command.

This example also shows the use of the reliable signals provided by 4.3BSD, which we described in Section 2.4.

Since a process can have only one signal handler for a given signal, if asynchronous I/O is enabled for more than one descriptor, the process doesn't know, when the signal occurs, which descriptor is ready for I/O. To do this, the `select` system call is used, which we describe in the next section.

6.13 Input/Output Multiplexing

Consider a process that reads input from more than one source. An actual example is the 4.3BSD line printer spooler that we cover in Section 13.2. It opens two sockets, a Unix stream socket to receive print requests from processes on the same host, and a TCP socket to receive print requests from processes on other hosts. Since it doesn't know when requests will arrive on the two sockets, it can't start a read on one socket, as it could block while waiting for a request on that socket, and in the mean time a request can arrive on the other socket. There are a few different techniques available to handle this multiplexing of different I/O channels.

1. It can set both sockets to nonblocking, using either the `FNDELAY` flag to `fcntl` or the `FIONBIO` request to `ioctl`. But the process has to execute a loop that reads from each socket, and if nothing is available to read, wait some amount of time before trying the reads again. This is called *polling*. The software polls each socket at some interval (every second, perhaps) and if no data is available to read, the process waits by calling the `sleep` function in the standard C library. Polling can waste computer resources, since most of the time there is no work to be done.
2. The process can `fork` one child process to handle each I/O channel. In this example it spawns two processes, one to read from the Unix socket and one to read from the TCP socket. Each child process calls `read` and blocks until data is available. When a child returns from its `read` it passes the data to the parent process using some form of IPC (that the parent must set up before spawning its children). Any of the techniques that we discussed in Chapter 3 can be used: pipes, FIFOs, message queues, shared memory with a semaphore, or another socket can also be used.
3. Asynchronous I/O can be used. The problem with this method is that signals are expensive to catch [Leffler et al. 1989]. Also, if more than one descriptor has been enabled for asynchronous I/O, the occurrence of the `SIGIO` signal doesn't tell us which descriptor is ready for I/O. Finally, as mentioned in Section 6.12, this technique is only available for terminals and sockets under 4.3BSD.

4. Another technique is provided by 4.3BSD with its `select` system call, which we describe below. This system call allows the user process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one of these events occurs. In this example the kernel can be instructed to notify the process only when data is available to be read from either of the two sockets that we're interested in.

The prototype for the system call is

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

int select(int maxfdpl, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

FD_ZERO(fd_set *fdset);              /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset);       /* turn the bit for fd on in fdset */
FD_CLR(int fd, fd_set *fdset);       /* turn the bit for fd off in fdset */
FD_ISSET(int fd, fd_set *fdset);     /* test the bit for fd in fdset */
```

The structure pointed to by the *timeout* argument is defined in `<sys/time.h>` as

```
struct timeval {
  long    tv_sec;     /* seconds */
  long    tv_usec;    /* microseconds */
};
```

The C `typedef` for the `fd_set` structure, and the definitions of the `FD_xxx` macros are in the `<sys/types.h>` include file. The definition of this system call makes it look more complicated than it is.

A request to `select` could be: tell us if any of the file descriptors in the set {1, 4, 5} are ready for reading, or if any of the file descriptors in the set {2, 7} are ready for writing, or if any of the file descriptors in the set {1, 4} have an exceptional condition pending. Additionally we can tell the kernel to

- Return immediately after checking the descriptors. This is a poll. For this, the *timeout* argument must point to a `timeval` structure, and the timer value (the number of seconds and microseconds specified by the structure) must be zero.
- Return when one of the specified descriptors is ready for I/O, but don't wait beyond a fixed amount of time. For this, the *timeout* argument must point to a `timeval` structure, and its value (the number of seconds and microseconds specified by the structure members) must be the nonzero amount of time to wait.
- Return only when one of the specified descriptors is ready for I/O—wait indefinitely. For this, the *timeout* argument must be NULL. This wait can also be interrupted by a signal.

Before describing how to specify the file descriptors to be checked, let's first show an example where we are not interested in any file descriptors. We'll use `select` as a higher precision timer than is provided by the `sleep` function. (`sleep` provides a resolution of seconds, whereas `select` allows us to specify a resolution in microseconds.)

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

main(argc, argv)
int     argc;
char    *argv[];
{
        long                    atol();
        static struct timeval   timeout;

        if (argc != 3)
                err_quit("usage: timer <#seconds> <#microseconds>");
        timeout.tv_sec  = atol(argv[1]);
        timeout.tv_usec = atol(argv[2]);

        if (select(0, (fd_set *) 0, (fd_set *) 0, (fd_set *) 0, &timeout) < 0)
                err_sys("select error");
        exit(0);
}
```

Here we have specified the second, third and fourth arguments to `select` as NULL.

The *readfds*, *writefds*, and *exceptfds* arguments specify which file descriptors we're interested in checking for each of the conditions—descriptor ready for reading, descriptor ready for writing and exceptional condition on descriptor, respectively. There are only two exceptional conditions currently supported.

1. The arrival of out-of-band data for a socket. We describe this in more detail in the next section.
2. The presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. We cover this in Section 15.10.

The problem the designers of this system call had was how to specify one or more descriptor values for each of these three arguments. The decision was made to represent the set of all possible descriptors using an array of integers, where each bit corresponds to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All this implementation detail is hidden through the `typedef` of the `fd_set` structure and the `FD_xxx` macros.

For example, to define a variable of type `fd_set` and then turn on the indicators for descriptors 1, 4, and 5

```
fd_set  fdvar;

FD_ZERO(&fdvar);        /* initialize the set - all bits off */
FD_SET(1, &fdvar);      /* turn on bit for fd 1 */
FD_SET(4, &fdvar);      /* turn on bit for fd 4 */
FD_SET(5, &fdvar);      /* turn on bit for fd 5 */
```

It is important to initialize the set, since unpredictable results can occur if the set is allocated as an automatic variable and not initialized. Any of the three arguments to `select`, *readfds*, *writefds*, or *exceptfds*, can be specified as NULL pointers, if we're not interested in that condition.

The *maxfdp1* argument specifies the number of file descriptors to be tested. Its value is the maximum file descriptor to be tested, plus one. The descriptors 0, 1, 2, up through and including *maxfdp1–1* are tested. The system allows for a potentially huge number of descriptors to be tested (currently 256), although most BSD kernels don't allow any single process to have that many open files. The *maxfdp1* argument lets us tell the system the largest descriptor that we're interested in, which is usually much less than 256. For example, given the example code above that turns on the indicators for descriptors 1, 4, and 5, a *maxfdp1* value of 6 can be specified. The reason it is 6 and not 5 is that we're specifying the number of descriptors, not the largest value, and descriptors start at zero.

This system call modifies the three arguments *readfds*, *writefds*, and *exceptfds* to indicate which descriptors are ready for the specified condition. These three arguments are value–result arguments. The caller should use the `FD_ISSET` macro to test a specific bit in an `fd_set` structure. The return value from this system call indicates the total number of descriptors that are ready. If the timer value expired before any of the descriptors were ready, a value of zero is returned. As usual, a return value of –1 indicates an error.

We have been talking about waiting for a file descriptor to become ready for I/O (reading or writing) or to have an exceptional condition pending on it. For sockets, this assumes that the process wants to do I/O on the socket. The `select` system call can also be used to await a connection on a socket. For example, if a process is awaiting a `connect` on more than one socket, if it doesn't want to execute the `accept` system call and potentially block, it can issue a `select` instead. If a `connect` is received by the system for a socket that is being waited for, the socket is then considered ready for reading, and the `select` returns.

6.14 Out-of-Band Data

We described the notion of out-of-band data in Chapter 4, and also mentioned it in Chapter 5 when we discussed the TCP and SPP protocols. Of all the protocols we've discussed so far, only TCP and SPP support out-of-band data. (Out-of-band data is only defined for stream sockets, and the Unix domain stream sockets don't support it.) Unfortunately, TCP and SPP treat out-of-band data differently. The designers of the socket interface tried to support out-of-band data in a generic way, but there are many programs that have been written using the TCP out-of-band data features that would be hard to port to SPP. The BSD socket abstraction calls for the protocol to support at least one outstanding out-of-band message at any time, and the out-of-band message must contain at least one byte of data.

To send an out-of-band message, the `MSG_OOB` flag must be specified for the `send`, `sendto`, or `sendmsg` system calls. SPP only allows a single byte of out-of-band data to be sent, while TCP has no restriction.

SPP Out-of-Band Data

When out-of-band data is received by the other end, several possibilities exist about how it is handled. Let's first consider how SPP handles it, as it is less complicated than the TCP implementation.

- The out-of-band byte is received along with a special flag specifying that the byte contains out-of-band data. (This flag is the Attention bit in the SPP header.)
- If the socket has a process group, the `SIGURG` signal is generated for the process group of the socket.
- The socket option `SO_OOBINLINE` has no effect, since the Xerox protocol specifies that the out-of-band byte is to be delivered again to the receiving process, in its normal position in the data stream.
- The process can read the out-of-band byte by calling any one of the three receive system calls, specifying the `MSG_OOB` flag. Only the single out-of-band byte is returned. If there is no out-of-band data when the `MSG_OOB` flag is specified, an error of `EINVAL` is returned by these three system calls instead.
- Regardless whether the process reads the out-of-band byte or not, the position of the byte in the normal data stream is remembered.
- The process continues reading data from the socket. Since the system remembers the position of the out-of-band data byte, it does not read right past it in a single read request. That is, if there are 20 bytes from the current read position until the out-of-band byte, and if we execute a read or receive system call specifying a length of 30 bytes, the system only returns 20 bytes. This forced stopping at the

out-of-band mark is to allow us to execute the SIOCATMARK ioctl to determine when we're at the mark. This allows us to ignore all the data up to the out-of-band byte. When this ioctl indicates that we're at the mark, the next byte we read is the out-of-band byte.

TCP Out-of-Band Data

All this gets more complicated with TCP because it can send the notification that out-of-band data exists (termed "urgent data" by TCP) *before* it sends the out-of-band data. In this case, if the process executes one of the three receive system calls, an error of EWOULDBLOCK is returned if the out-of-band data has not arrived. As with the SPP example above, an error of EINVAL is returned by these three system calls if the MSG_OOB flag is specified when there isn't any out-of-band data.

Still another option is provided with the TCP implementation, to allow for multiple bytes of out-of-band data. By default, only a single byte of out-of-band data is provided, and unlike SPP, this byte of data is not stored in the normal data stream. This data byte can only be read by specifying the MSG_OOB flag to the receive system calls. But if we set the SO_OOBINLINE option for the socket, using the setsockopt system call described in Section 6.11, the out-of-band data is left in the normal data stream and is read without specifying the MSG_OOB flag. If this socket option is enabled, we must use the SIOCATMARK ioctl to determine where in the data stream the out-of-band data occurs. In this case, if multiple occurrences of out-of-band data are received, all the data is left in-line (i.e., none of the data can be lost) but the mark returned by the SIOCATMARK ioctl corresponds to the final sequence of out-of-band data that was received. If the out-of-band data is not received in-line, it is possible to lose some intermediate out-of-band data when the out-of-band data arrives faster than it is processed by the user.

The remote login application in Chapter 15 uses TCP out-of-band data along with the SIOCATMARK ioctl.

6.15 Sockets and Signals

We have mentioned signals in relation to sockets a few times in this chapter, and its worth taking a moment to summarize the conditions under which signals are generated for a socket and all the related nuances. There are three signals that can be generated for a socket

SIGIO This signal indicates that a socket is ready for asynchronous I/O. The signal is sent to the process group of the socket. This process group is established by calling ioctl with a command of either FIOSETOWN or SIOCSPGRP, or by calling fcntl with a command of F_SETOWN. This

signal is sent to the process group only if the process has enabled asynchronous I/O on the socket by calling `ioctl` with a command of `FIOASYNC` or by calling `fcntl` with a command of `F_SETFL` and an argument of `FASYNC`.

`SIGURG` This signal indicates that an urgent condition is present on a socket. An urgent condition is either the arrival of out-of-band data on the socket or the presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. (We discuss pseudo-terminal packet mode in Section 15.10.) The signal is sent to the process group of the socket. This process group is established by calling `ioctl` with a command of either `FIOSETOWN` or `SIOCSPGRP`, or by calling `fcntl` with a command of `F_SETOWN`.

`SIGPIPE` This signal indicates that we can no longer write to a socket, pipe, or FIFO. Nothing special is required by a process to receive this signal, but unless the process arranges to catch the signal, the default action is to terminate the process. This signal is sent only to the process associated with the socket; the process group of the socket is not used for this signal.

We mentioned in Section 2.4 that certain system calls (termed the "slow" system calls) can be interrupted when the process handles a signal. This always has to be considered when handling signals. Although 4.3BSD tries automatically to restart certain system calls that are interrupted, the `accept` and `recvfrom` system calls are never restarted automatically by the kernel. Since both of these system calls can typically block, if you are using them and handling signals, be prepared to restart the system call if they're interrupted.

6.16 Internet Superserver

On a typical 4.3BSD system there can be many daemons in existence, just waiting for a request to arrive. For example, there could be an Internet FTP daemon, an Internet TELNET daemon, and Internet TFTP daemon, a remote login daemon, a remote shell daemon, and so on. With systems before 4.3BSD, each of these services had a process associated with it. This process was started at boot time from the `/etc/rc` startup file, and each process did nearly identical startup tasks: create a socket, `bind` the server's well-known address to the socket, wait for a connection, then `fork`. The child process performed the service while the parent waited for another request.

The 4.3BSD release simplified this by providing an Internet superserver: the `inetd` process. This daemon can be used by a server that uses either TCP or UDP. It does not handle either the XNS or Unix domain protocols. What this daemon provides is two features:

- It allows a single process (`inetd`) to be waiting to service multiple connection requests, instead of one process for each potential service. This reduces the total number of processes in the system.
- It simplifies the writing of the daemon processes to handle the requests, since many of the start-up details are handled by `inetd`. This obviates the need for the actual service process to call the `daemon_start` function that we developed in Section 2.6. All the details that we said had to be handled by a typical daemon, are done by `inetd`, before the actual server is invoked.

There is a price to pay for this, however, in that the `inetd` daemon has to execute both a `fork` and an `exec` to invoke the actual server process, while a self-contained daemon that did everything itself only has to execute a `fork` to handle each request. This additional overhead, however, is worth the simplification of the actual servers and the reduction in the total number of processes in the system.

The `inetd` process establishes itself as a daemon using many of the techniques that we described in Section 2.6. It then reads the file `/etc/inetd.conf` to initialize itself. This text file specifies the services that the superserver is to listen for, and what to do when a service request arrives. Each line contains the fields shown in Figure 6.17.

Field	Description
service-name	must be in `/etc/services`
socket-type	`stream` or `dgram`
protocol	must be in `/etc/protocols`: either `tcp` or `udp`
wait-flag	`wait` or `nowait`
login-name	from `/etc/passwd`: typically `root`
server-program	full pathname to `exec`
server-program-arguments	maximum of 5 arguments

Figure 6.17 Fields in `inetd` configuration file.

Some sample lines are

```
ftp     stream  tcp     nowait  root    /etc/ftpd       ftpd
telnet  stream  tcp     nowait  root    /etc/telnetd    telnetd
login   stream  tcp     nowait  root    /etc/rlogind    rlogind
tftp    dgram   udp     wait    nobody  /etc/tftpd      tftpd
```

The actual name of the server is always passed as the first argument to a program when it is `exec`ed.

A picture of what the daemon does is shown in Figure 6.18. Let's go through a typical scenario for this daemon.

1. On startup it reads the `/etc/inetd.conf` file and creates a socket of the appropriate type (stream or datagram) for all the services specified in the file. Realize that

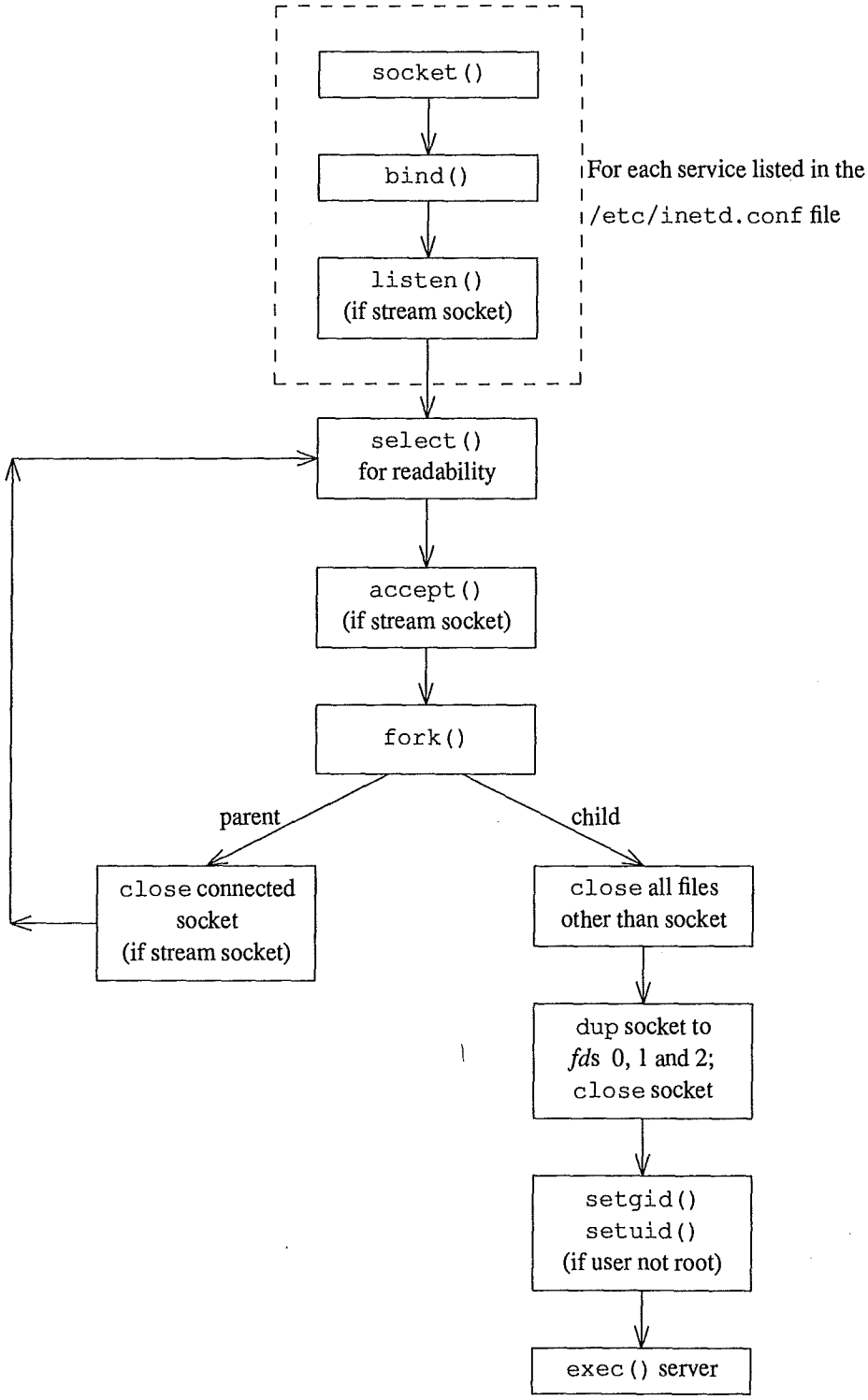


Figure 6.18 Steps performed by `inetd`.

there is a limit to the number of servers that `inetd` can be waiting for, as the total number of file descriptors used by `inetd` can't exceed the system's per-process limit. On most BSD systems this limit is 64, but it can be changed by reconfiguring the kernel.

2. As each socket is created, a `bind` is executed for every socket, specifying the well-known address for the server. This TCP or UDP port number is obtained by looking up the *service-name* field from the configuration file in the `/etc/services` file. Some typical lines in this file, corresponding to the example lines in the configuration file that we showed above, are

```
ftp         21/tcp
telnet      23/tcp
tftp        69/udp
login       513/tcp
```

   Both the *service-name* (such as `telnet`) and the *protocol* from the `inetd` configuration file are passed as arguments to the library function `getservbyname`, to locate the correct port number for the `bind`. (We describe the `getservbyname` function in Section 8.2.)

3. For stream sockets, a `listen` is executed, specifying a willingness to receive connections on the socket and the queue length for incoming connections. This step is not done for datagram sockets.

4. A `select` is then executed, to wait for the first socket to become ready for reading. Recall from our description of this system call in Section 6.13, that a stream socket is considered ready for reading when a connection request arrives for that socket. A datagram socket is ready for reading when a datagram arrives. At this point the `inetd` daemon just waits for the `select` system call to return.

5. When a socket is ready for reading, if it is a stream socket, an `accept` system call is executed to accept the connection.

6. The `inetd` daemon `forks` and the child process handles the service request. The child closes all the file descriptors other than the socket descriptor that it is handling and then calls `dup2` to cause the socket to be duplicated on file descriptors 0, 1, and 2. The original socket descriptor is then closed. Doing this, the only file descriptors that are open in the child are 0, 1, and 2. It then calls `getpwnam` to get the password file entry for the *login-name* that is specified in the `/etc/inetd.conf` file. If this entry does not have a user ID of zero (the superuser) then the child becomes the specified user by executing the `setgid` and `setuid` system calls. (Since the `inetd` process is executing with a user ID of zero, the child process inherits this user ID across the `fork`, so it is able to become any user that it chooses.) The child process now does an `exec` to execute the appropriate *server-program* to handle the request, passing the arguments that are specified in the configuration file.

7. If the socket is a stream socket, the parent process must close the connected socket. The parent goes back and executes the `select` system call again, waiting for the next socket to become ready for reading.

The scenario we've described above handles the case where the configuration file specifies `nowait` for the server. This is typical for all TCP services. If another connection request arrives for the same server, it is returned to the parent process as soon as it executes the `select`. The steps listed above are executed again, and another child process handles this new service request.

Specifying the `wait` flag for a datagram service changes the steps done by the parent process. First, after the `fork` the parent saves the process ID of the child, so it can tell later when that specific child process terminates, by looking at the value returned by the `wait` system call. Second, the parent disables the current socket from future `selects` by using the `FD_CLR` macro to turn off the bit in the `fd_set` structure that it uses for the `select`. This means that the child process takes over the socket until it terminates. Once the child process terminates, the parent process is notified by a `SIGCLD` signal, and the parent's signal handler obtains the process ID of the terminating child and reenables the `select` for the corresponding socket by using the `FD_SET` macro. When the child process terminates, the parent is probably waiting for its `select` system call to return. As mentioned in Section 2.4, when a signal handler is invoked while a process is executing a "slow" system call, when the signal handler returns, the system call (the `select` in this case) returns with an error indication and an `errno` value of `EINTR`. The `inetd` process recognizes this and executes the `select` system call again. But by executing the system call again, it can now wait for the socket corresponding to the child process that terminated. If the occurrence of the signal did not interrupt the `select` system call, the parent would not be able to wait for the socket corresponding to the terminating child process. When we go through the TFTP example in Chapter 12 we'll see how a datagram server uses the `wait` mode.

It is worth going through a time sequence of the steps involved in the previous paragraph, making certain we understand the interaction of the parent and child processes, the `SIGCLD` signal, and how the `select` system call can be interrupted.

- The datagram request arrives on socket *N*, the `select` returns to the `inetd` process.
- A child process is `forked` and `execed` to handle the request.
- `inetd` disables socket descriptor *N* from its `fd_set` structure for the `select`. The child process takes over the socket.
- The child handles this request and `inetd` handles requests for other services.
- Eventually `inetd` calls `select` and blocks.
- The child terminates, the `SIGCLD` signal is generated for the `inetd` process.

- `inetd` handles the signal and obtains the process ID of the terminating child process from the `wait` system call. It figures out which socket descriptor corresponded to this child process and turns on the appropriate bit in its `fd_set` structure.
- o When the signal handler returns, the `select` returns to the `inetd` process, with an `errno` of `EINTR`.
- `inetd` calls `select` again, this time with an `fd_set` structure that enables socket descriptor *N*.

While the `inetd` daemon is set up to handle both a TCP socket with the `wait` flag and a UDP socket with the `nowait` flag, there are no examples of this in the distributed 4.3BSD system.

The `inetd` daemon has several servers that are handled by the daemon itself. These internal servers handle some of the standard Internet services.

- Echo service—RFC 862 [Postel 1983a]
- Discard service—RFC 863 [Postel 1983b]
- Character generator service—RFC 864 [Postel 1983c]
- Daytime (human readable) service—RFC 867 [Postel 1983d]
- Machine time (binary) service—RFC 868 [Postel and Harrenstien 1983]

Each of these services can be contacted using either TCP or UDP. The TCP servers for these internal functions are handled as iterative servers if the amount of time to handle the request is fixed (the daytime and machine time servers), or as concurrent servers if the amount of time depends on the request (echo, discard and character generator).

Since `inetd` is the process that does the `accept` of a stream connection, the server that is invoked by `inetd` has to execute the `getpeername` system call to obtain the address of the client. This is done, for example, by servers that want to verify that the client is using a reserved port. For datagram servers, the address of the client is returned to the server when it executes one of the receive system calls.

6.17 Socket Implementation

Sockets, as implemented by 4.3BSD, are implemented within the Unix kernel. All the system calls that we discussed in this chapter are entry points into the kernel. All the algorithms and code to support the communication protocols (TCP, UDP, IDP, etc.) are completely within the kernel. Adding a new protocol (such as the OSI-related standards) requires changing the kernel. The overall networking design gets more modular with each release (the addition of XNS support in 4.3BSD required changes, as documented in O'Toole, Torek, and Weiser [1985], and the addition of OSI protocols in 4.4BSD will require additional changes that weren't anticipated in earlier releases), but it still requires kernel changes.

The actual kernel implementation separates the network system into three layers, as shown in Figure 6.19. See Leffler et al. [1989] for additional details.

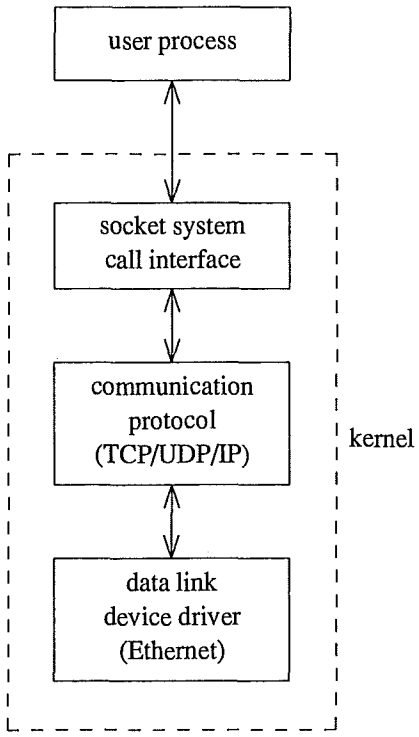


Figure 6.19 Networking implementation in 4.3BSD.

6.18 Summary

This has been a long chapter with many details. Our presentation was divided into four parts.

- Elementary socket system calls
- Examples using only the elementary socket system calls
- Advanced socket system calls
- Advanced socket features and options

One key point is that the simple examples presented in Section 6.6 can form the basis for any networking application. Just modify the client and server to do whatever you desire.

We'll present additional features for these examples in Chapter 8. For example, in Section 8.2 we show a better way of obtaining the address of a remote system (instead of hard coding it in a header file). Section 8.3 develops some functions that simplify the standard operations that most client applications execute when opening a socket. Section 8.4 shows how to add reliability to a datagram application.

The advanced features that we've described in this chapter are used in the examples that follow in the text. For example, the remote login example in Chapter 15 uses socket options, I/O multiplexing, out-of-band data and signals. You should review the later sections of this chapter as you encounter examples that use a particular feature.

Exercises

6.1 The example servers in Section 6.6 do not call the `daemon_start` function which we developed in Section 2.6. Modify one of these servers to call this function. What other changes do you have to make to the server?

6.2 Rewrite the SPP example from Section 6.6 to use the packet-stream interface, `SOCK_SEQPACKET`, and set the end-of-message bit at the end of every line.

6.3 What happens with the clients and servers in Section 6.6 if standard input is a binary file?

6.4 Code a function that is similar to `rresvport`, but for UDP sockets.

6.5 Print the value of `TCP_MAXSEG` after the socket has been connected to a peer process. If possible, connect to a peer process on a LAN and print value and then connect to a peer process on a WAN and compare the results.

6.6 Write a program that copies standard input to standard output using the `select` system call.

6.7 Now that we have covered asynchronous I/O and I/O multiplexing, compare the use of Unix domain stream sockets and message queues. If you had to write a server to handle multiple client requests on a single host, which would you use and why?

6.8 Write a simple client and server to verify that a single datagram is produced by the `writev` system call.

6.9 If your system supports file access across a network (such as Sun's NFS) test the Unix stream example from Section 6.6 with the client and server on different systems. Modify the pathname used by the client and server to be on a filesystem that is accessible to both.