NEAR EAST UNIVERSITY

Faculty of Engineering

Department Of Computer Engineering

TRANSMISSION CONTROL PROTOCOL – USER DATAGRAM PROTOCOL (TCP-UDP)

Graduation Project
COM – 400

Student : Asif Khurshid

Supervisor: Assist. Prof. Dr. Firudin Muradov

Nicosia – 2003

ACKNOWLEDGEMENTS

"All glory to Almighty ALLAH, the Lord of universe, Who is the entire source of all knowledge and wisdom endowed to mankind. All thanks are due to ALLAH who enabled me to complete this project.

I would like to say special thanks to my project advisor Dr Firudin Muradov for his deep interest, continuous guidance, assistance and cooperation at every stage of the project.

Then I want to say thanks to my family for their encouragement and support.

Finally I would like to thank all of my friends for their help."

ABSTRACT

Transmission Control Protocol/Internet Protocol (TCP/IP) is an industry-standard suit of protocols designed for Wide Area Networks (WANs). The roots of the TCP/IP can be traced back to the packet switching network experiments conducted by the US Department of Defense Advanced Research Projects Agency (DARPA). IP is a connectionless protocol primarily responsible for addressing and routing packets between hosts, that is, a session is not established before exchanging data. IP is unreliable in that delivery is not guaranteed. An acknowledgement is not required when data is received. Where as Transmission Control Protocol (TCP) is responsible for controlling the transmission of data from one host to another host. The TCP/IP utilities include File Transfer Protocol (FTP), Trivial File Transfer Protocol (TFTP), Remote Copy Protocol (RCP), Telnet, Remote Shell (RSH), Remote Execution (REXEC), Line Printer Remote (LPR), Line Printer Queue (LPQ), Line Printer Daemon (LPD).

User Datagram Protocol (UDP) is a connectionless protocol used with both the Trivial File Transfer Protocol (TFTP) and the Remote Call Procedure (RCP). Connectionless communications don't provide reliability, meaning there is no indication to the sending device that a message has been received correctly. It also does not gives error recovery facility like IP.

TABLE 0F CONTENTS

ACKNOWLEDEGMENT i

ABSTRACT ii

TABLE OF CONTENTS iii

INTRODUCTION 1

CHAPTER ONE: OPEN SYSTEMS, STANDARDS, AND PROTOCOLS 3

1.1 Open Systems 3

1.1.1 What Is an Open System? 4

1.2 Network Architectures 6

1.2.1 Local Area Networks 7

1.2.1.1 The Bus Network 7

1.2.1.2 The Ring Network 10

1.2.1.3 The Hub Network 10

1.2.2 Wide Area Networks 11

1.3 Layers 13

1.3.1 The Application Layer 14

1.3.2 The Presentation Layer 15

1.3.3 The Session Layer 15

1.3.4 The Transport Layer 15

1.3.5 The Network Layer 16

1.3.6 The Data Link Layer 16

1.3.7 The Physical Layer 16

1.4 Terminology and Notations 17

1.4.1 Packets 17

1.4.2 Subsystems 17

1.4.3 Entities 18

1.4.4 N Notation 18

1.4.5 N-Functions 18

1.4.6 N-Facilities 18

1.4.7 Services 18

1.4.8 Making Sense of the Jargon 20

1.4.9 Queues and Connections 21

1.5 Standards 23

1.5.1 Setting Standards 23

1.5.2 Internet Standards 24

1.6 Protocols 26

1.6.1 Breaking Data Apart 27

1.6.2 Protocol Headers 29

CHAPTER TWO: TCP and UDP 32

2.1 What Is TCP? 32

2.2 Following a Message 34

2.3 Ports and Sockets 36

2.4 TCP Communications with the Upper Layers 41

2.5 Passive and Active Ports 43

2.6 TCP Timers 44

2.6.1 The Retransmission Timer 44

2.6.2 The Quiet Timer 45

2.6.3 The Persistence Timer 45

2.6.4 The Keep-Alive Timer and the Idle Timer 45

2.7 Transmission Control Blocks and Flow Control 45

2.8 TCP Protocol Data Units 47

2.9 TCP and Connections 49

2.9.1 Establishing a Connection 49

2.9.2 Data Transfer 51

2.9.3 Closing Connections 52

2.10 User Datagram Protocol (UDP) 54

CHAPTER THREE: TCP/UDP and Networks 56

3.1 TCP/UDP and Other Protocols 56

3.1.1 LAN Layers 57

3.1.2 NetBIOS and TCP/IP 59

3.1.3 XNS and TCP/IP 61

3.1.4 IPX and UDP 61

3.1.5 ARCnet and TCP/IP 62

3.1.6 FDDI Networks 62

3.1.7 X.25 and IP 63

3.1.8 ISDN and TCP/IP 63

3.1.9 Switched Multi-Megabit Data Services and IP 64

3.1.10 Asynchronous Transfer Mode (ATM) and BISDN 64

3.1.11 Windows 95 and TCP/IP 64

3.2 Optional TCP/UDP Services 66

3.2.1 Active Users 67

3.2.2 Character Generator 67

3.2.3 Daytime 68

3.2.4 Discard 68

3.2.5 Echo 68

3.2.6 Quote of the Day 68

3.2.7 Time 69

3.2.8 Using the Optional Services 69

3.3 Setting Up a Sample TCP/IP Network: Servers 71

3.3.1 The Sample Network 71

3.3.2 Configuring TCP/IP Software 72

3.4 Setting Up a Sample TCP/IP Network: DOS and Windows Clients 78

3.4.1 DOS-Based TCP/IP: ftp Software's PC/TCP 78

3.4.1.1 Installing PC/TCP 80

3.4.1.1.1 The AUTOEXEC.BAT File 80

3.4.1.1.2 The CONFIG.SYS File 83

3.4.1.1.3 The PROTOCOL.INI File 85

3.4.1.1.4 The PCTCP.INI File 86

3.4.1.1.5 The Windows SYSTEM.INI File 87

3.4.1.2 Windows for Workgroups using NetBIOS 89

3.4.1.3 Testing PC/TCP 92

CHAPTER FOUR: WINSOCK AND THE SOCKET PROGRAMMING INTERFACE 98

4.1 Winsock 98

4.1.1 Trumpet Winsock 98

4.1.2 Installing Trumpet Winsock 99

4.1.3 Configuring the TCP/IP Packet Driver 99
4.2 The Socket Programming Interface 101
4.2.1 Development of the Socket Programming Interface 101
4.2.2 Socket Services 102
4.2.2.1 Transmission Control Block 103
4.2.2.2 Creating a Socket 103
4.2.2.3 Binding the Socket 103
4.2.2.4 Connecting to the Destination 105
4.2.2.5 The *open* Command 105
4.2.2.6 Sending Data 107
4.2.2.7 Receiving Data 109
4.2.2.8 Server Listening 110
4.2.2.9 Getting Status Information 112
4.2.2.10 Closing a Connection 113
4.2.2.11 Aborting a Connection 114
4.2.2.12 UNIX Forks 114
CONCLUSION **115**
REFERENCES **116**

INTRODUCTION

The internet consists of thousands of network wordwide connecting research facilities, universities, libraries, government agencies and private companies. TCP/UDP are the standard, routable entries networking protocols. All modern operating systems offer TCP support and most large networks rely on TCP for much of their network traffic. This is a technology for connecting dissimilar systems.

In the first chapter, we will see the Open Systems, what is an open system, Network Structures, local area networks such as the bus network, the ring network, the hub network, wide area networks, Layers such as the application layer, the presentation layer, the session layer, the transport layer, the network layer, the data link layer, the physical layer, Standards such as setting standards, internet standards, Protocols, breaking data apart, and protocol headers.

The second chapter begins with definition of TCP/IP. The rest of chapter covers Following a Message, Ports and Sockets, TCP Communications with Upper Layers, Passive and Active Ports, TCP Timers, the retransmission timer, the quiet timer, the persistence timer, the keep-alive timer and the idle timer, Transmission Control Blocks and Flow Control, TCP Protocols Data Units, TCP and Connections, establishing a connection, data transfer, closing connections, and UDP(User Datagram Protocol).

The third chapter covers the TCP-UDP and Networks, including TCP-UDP and Other Protocols, LAN layers, NetBIOS and TCP/IP, XNS and TCP/IP, IPX and UDP, ARCnet and TCP/IP, FDDI Networks, X.25 and IP, ISDN and TCP/IP, Switched Multi-Megabit Data Services and IP, Asynchronous Transfer Mode (ATM) and BISDN, Optional TCP/UDP services, active users, character generator, daytime, discard, echo, quote of the day, time, using the optional services, Setting Up a Sample TCP/IP Network Servers, the sample network, configuring TCP/IP software, and Setting Up a Sample TCP/IP Network: DOS and Windows Clients.

In the fourth chapter, we will see the Winsock and the Socket Programming Interface, including Trumpet Winsock, Installing Trumpet Winsock, Configuring the

TCP/IP Packet Driver, Socket Services, transmission control block, creating a socket, binding the socket, connecting to the destination, the open command, sending data, receiving data, server listening, getting status information, closing a connection, aborting a connection, and UNIX Forks.

CHAPTER ONE

OPEN SYSTEMS, STANDARDS AND PROTOCOLS

This chapter covers some important information, including the following:

- What an open system is
- How an open system handles networking
- Why standards are required
- How standards for protocols like TCP/IP are developed
- What a protocol is
- The OSI protocols

1.1 Open Systems

Primarily because TCP/IP grew out of the need to develop a standardized communications procedure that would inevitably be used on a variety of platforms. The need for a standard, and one that was readily available to anyone (hence *open*), was vitally important to TCP/IP's success. Therefore, a little background helps put the design of TCP/IP into perspective.

More importantly, open systems have become de rigueur in the current competitive market. The term *open system* is bandied around by many people as a solution for all problems (to be replaced occasionally by the term *client/server*), but neither term is usually properly used or understood by the people spouting them. Understanding what an open system really is and what it implies leads to a better awareness of TCP/IP's role on a network and across large internetworks like the Internet.

In a similar vein, the use of standards ensures that a protocol such as TCP/IP is the same on each system. This means that our PC can talk to a minicomputer running TCP/IP without special translation or conversion routines. It means that an entire network of different hardware and operating systems can work with the same network protocols. Developing a standard is not a trivial process. Often a single standard involves more than a single document describing a software system. A standard often involves the interrelationship of many different protocols, as does TCP/IP. Knowing the interactions between TCP/IP and the other components of a

communications system is important for proper configuration and optimization, and to ensure that all the services we need are available and interworking properly.

1.1.1 What Is an Open System?

An open system is best loosely defined as one for which the architecture is not a secret. The description of the architecture has been published or is readily available to anyone who wants to build products for a hardware or software platform. This definition of an open system applies equally well to hardware and software.

A decade ago, open systems were virtually nonexistent. Each hardware manufacturer had a product line, and we were practically bound to that manufacturer for all our software and hardware needs. Some companies took advantage of the captive market, charging outrageous prices or forcing unwanted configurations on their customers. The groundswell of resentment grew to the point that customers began forcing the issue. The lack of choice in software and hardware purchases is why several dedicated minicomputer and mainframe companies either went bankrupt or had to accept open system principles: their customers got fed up with relying on a single vendor. A good example of a company that made the adaptation is Digital Equipment Corporation (DEC). They moved from a proprietary operating system on their VMS minicomputers to a UNIX-standard open operating system. By doing that, they kept their customers happy, and they sold more machines. That's one of the primary reasons DEC is still in business today.

UNIX is a classic example of an open software platform. UNIX has been around for 30 years. The source code for the UNIX operating system was made available to anyone who wanted it, almost from the start. UNIX's source code is well understood and easy to work with, the result of 30 years of development and improvement. UNIX can be ported to run on practically any hardware platform, eliminating all proprietary dependencies. The attraction of UNIX is not the operating system's features themselves but simply that a UNIX user can run software from other UNIX platforms, that files are compatible from one UNIX system to another (except for disk formats), and that a wide variety of vendors sell products for UNIX.

The growth of UNIX pushed the large hardware manufacturers to the open systems principle, resulting in most manufacturers licensing the right to produce a UNIX version for their own hardware. This step let customers combine different

hardware systems into larger networks, all running UNIX and working together. Users could move between machines almost transparently, ignorant of the actual hardware platform they were on. Open systems, originally of prime importance only to the largest corporations and governments, is now a key element in even the smallest company's computer strategy.

The term *open system networking* means many things, depending on whom we ask. In its broadest definition, open system networking refers to a network based on a well-known and understood protocol (such as TCP/IP) that has its standards published and readily available to anyone who wants to use them. Open system networking also refers to the process of networking open systems (machaine-specific hardware and software) using a network protocol. It is easy to see why people want open systems networking, though. Three services are widely used and account for the highest percentage of network traffic: file transfer, electronic mail, and remote login. Without open systems networking, setting up any of these three services would be a nightmare.

File transfers enable users to share files quickly and efficiently, without excessive duplication or concerns about the transport method. Network file transfers are much faster than an overnight courier crossing the country, and usually faster than copying a file on a disk and carrying it across the room. File transfer is also extremely convenient, which not only pleases users but also eliminates time delays while waiting for material. A common open system governing file transfers means that any incompatibilities between the two machines transferring files can be overcome easily.

Electronic mail has mushroomed to a phenomenally large service, not just within a single business but worldwide. The Internet carries millions of messages from people in government, private industry, educational institutions, and private interests. Electronic mail is cheap (no paper, envelope, or stamp) and fast (around the world in 60 seconds or so). It is also an obvious extension of the computer-based world we work in. Without an open mail system, we wouldn't have anywhere near the capabilities we now enjoy.

Finally, remote logins enable a user who is based on one system to connect through a network to any other system that accepts him as a user. This can be in the next workgroup, the next state, or in another country. Remote logins enable users to

take advantage of particular hardware and software in another location, as well as to run applications on another machine. Once again, without an open standard, this would be almost impossible.

1.2 Network Architectures

The term *network* usually means a set of computers and peripherals (printers, modems, plotters, scanners, and so on) that are connected together by some medium. The connection can be direct (through a cable) or indirect (through a modem). The different devices on the network communicate with each other through a predefined set of rules (the protocol).

The devices on a network can be in the same room or scattered through a building. They can be separated by many miles through the use of dedicated telephone lines, microwave, or a similar system. They can even be scattered around the world, again connected by a long-distance communications medium. The layout of the network (the actual devices and the manner in which they are connected to each other) is called the *network topology.*

Usually, if the devices on a network are in a single location such as a building or a group of rooms, they are called a local area network, or LAN. LANs usually have all the devices on the network connected by a single type of network cable. If the devices are scattered widely, such as in different buildings or different cities, they are usually set up into several LANs that are joined together into a larger structure called a wide area network, or WAN. A WAN is composed of two or more LANs. Each LAN has its own network cable connecting all the devices in that LAN. The LANs are joined together by another connection method, often high-speed telephone lines or very fast dedicated network cables called backbones.

One last point about WANs: they are often treated as a single entity for organizational purposes. For example, the ABC Software company might have branches in four different cities, with a LAN in each city. All four LANs are joined together by high-speed telephone lines. However, as far as the Internet and anyone outside the ABC Software company are concerned, the ABC Software WAN is a single entity. (It has a single domain name for the Internet.)

1.2.1 Local Area Networks

TCP/IP works across LANs and WANs, and there are several important aspects of LAN and WAN topologies we should know about. We can start with LANs and look at their topologies. Although there are many topologies for LANs, three topologies are dominant: bus, ring, and hub.

1.2.1.1 The Bus Network

The bus network is the simplest, comprising a single main communications pathway with each device attached to the main cable (bus) through a device called a transceiver or junction box. The bus is also called a backbone because it resembles a human spine with ribs emanating from it. From each transceiver on the bus, another cable (often very short) runs to the device's network adapter. An example of a bus network is shown in Figure 1.1.

The primary advantage of a bus network is that it allows for a high-speed bus. Another advantage of the bus network is that it is usually immune to problems with any single network card within a device on the network. This is because the transceiver allows traffic through the backbone whether a device is attached to the junction box or not. Each end of the bus is terminated with a block of resistors or a similar electrical device to mark the end of the cable electrically. Each device on the pathway has a special identifying number, or address, that lets the device know that incoming information is for that device.

A bus network is seldom a straight cable. Instead, it is usually twisted around walls and buildings as needed. It does have a single pathway from one end to the other, with each end terminated in some way (usually with a resistor). Figure 1.1 shows a logical representation of the network, meaning it has simplified the actual physical appearance of the network into a schematic with straight lines and no real scale to the connections. A physical representation of the network would show how it goes through walls, around desks, and so on. Most devices on the bus network can send or receive data along the bus by packaging a message with the intended recipient's address.

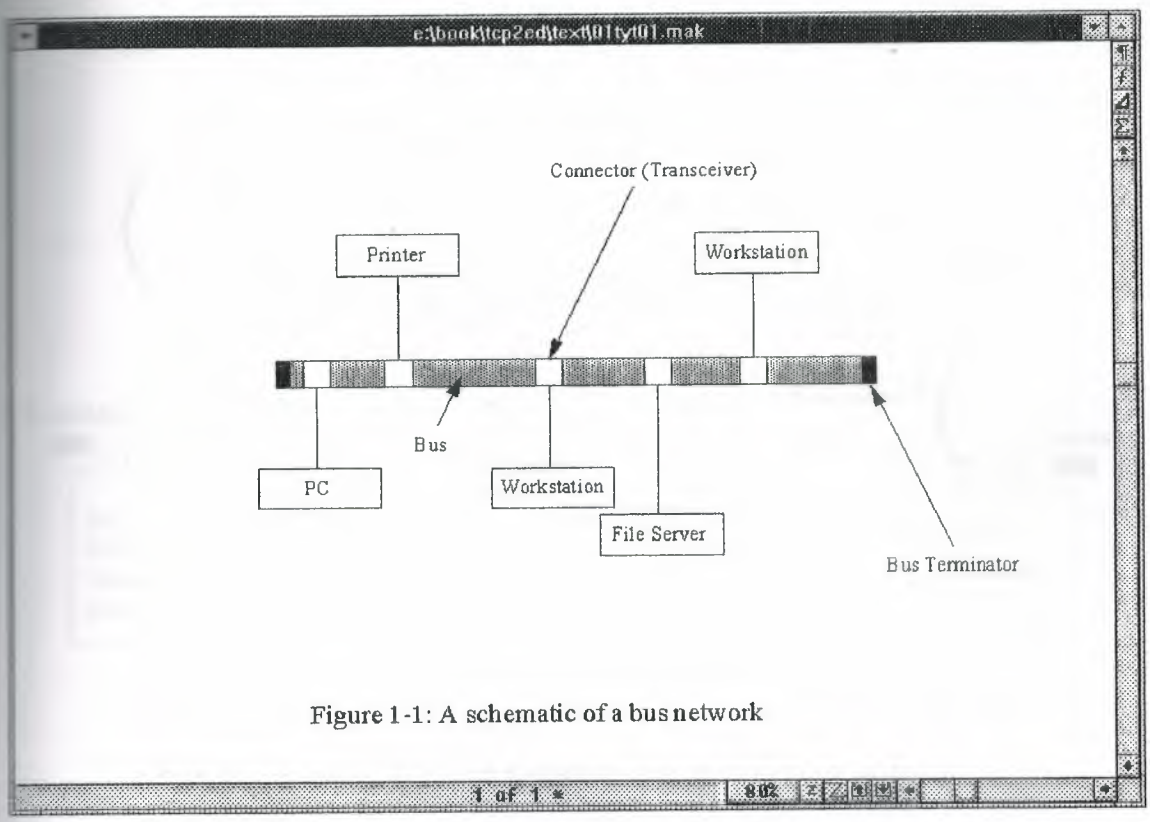

Figure 1-1: A schematic of a bus network

Figure 1.1 A schematic of a bus network showing the backbone with transceivers leading to network devices.

A variation of the bus network topology is found in many small LANs that use Thin Ethernet cable (which looks like television coaxial cable) or twisted-pair cable (which resembles telephone cables). This type of network consists of a length of coaxial cable that snakes from machine to machine. Unlike the bus network in Figure 1.1, there are no transceivers on the bus. Instead, each device is connected into the bus directly using a T-shaped connector on the network interface card, often using a connector called a BNC. The connector connects the machine to the two neighbors through two cables, one to each neighbor. At the ends of the network, a simple resistor is added to one side of the T-connector to terminate the network electrically.

A schematic of this type of network is shown in Figure 1.2. Each network device has a T-connector attached to the network interface card, leading to its two neighbors. The two ends of the bus are terminated with resistors.

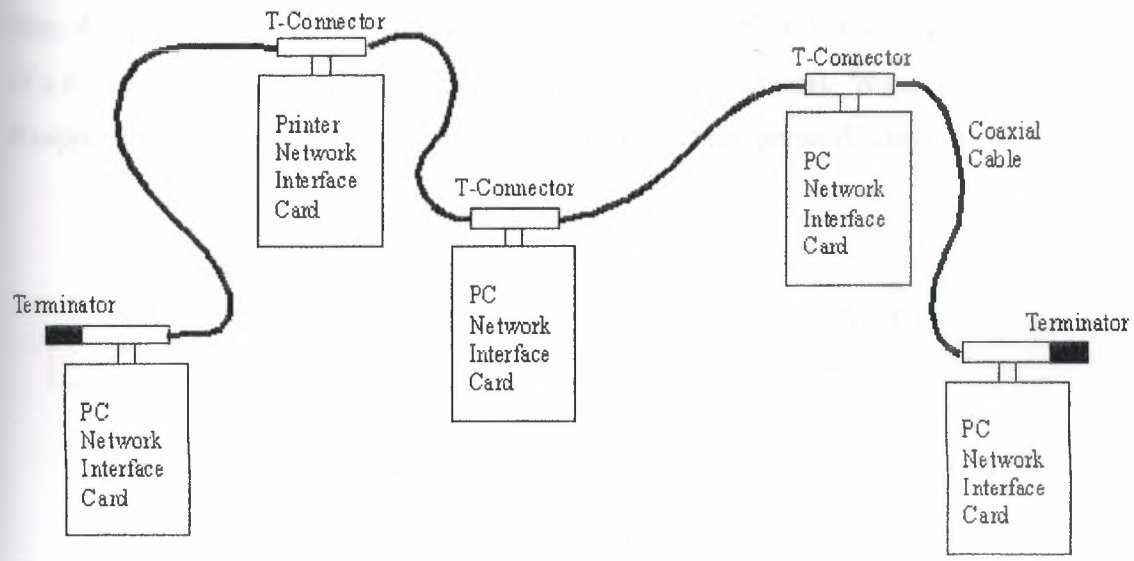

Figure 1.2 A schematic of a machine-to-machine bus network.

This machine-to-machine (also called peer-to-peer) network is not capable of sustaining the higher speeds of the backbone-based bus network, primarily because of the medium of the network cable. A backbone network can use very high-speed cables such as fiber optics, with smaller (and slower) cables from each transceiver to the device. A machine-to-machine network is usually built using twisted-pair or coaxial cable because these cables are much cheaper and easier to work with. Until recently, machine-to-machine networks were limited to a throughput of about 10 Mbps (megabits per second), although recent developments called 100VG AnyLAN and Fast Ethernet allow 100 Mbps on this type of network.

The advantage of this machine-to-machine bus network is its simplicity. Adding new machines to the network means installing a network card and connecting the new machine into a logical place on the backbone. One major advantage of the machine-to-machine bus network is also its cost: it is probably the lowest cost LAN topology available. The problem with this type of bus network is that if one machine is taken off the network cable, or the network interface card malfunctions, the backbone is broken and must be tied together again with a jumper of some sort or the network might cease to function properly.

1.2.1.2 The Ring Network

A ring network topology is often drawn as its name suggests, shaped like a ring. A typical ring network schematic is shown in Figure 1.3. We might have heard of a *token ring network* before, which is a ring topology network. We might be disappointed to find no physical ring architecture in a ring network, though.

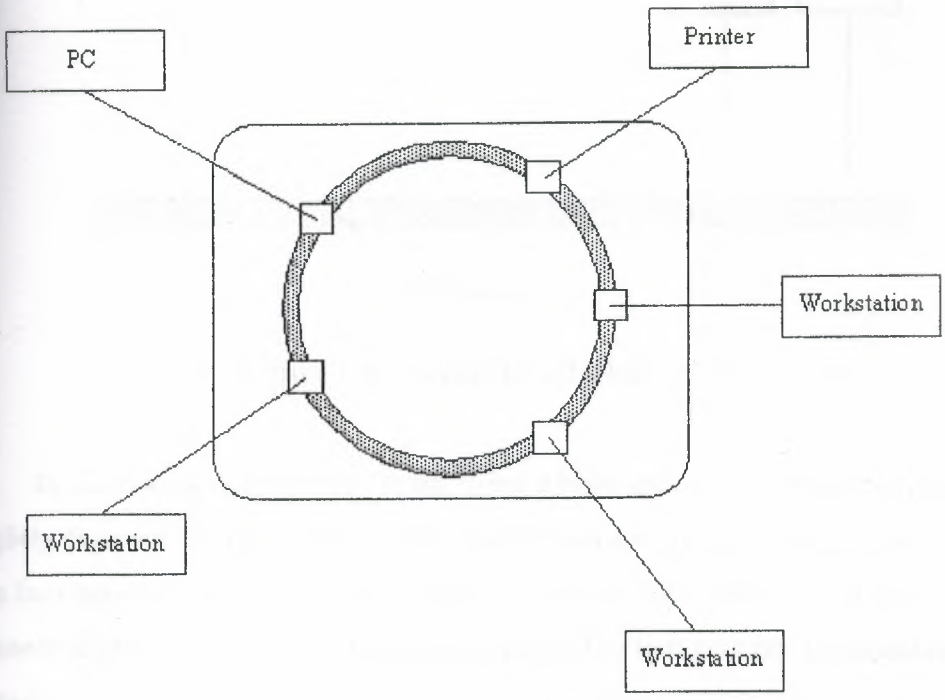

Figure 1.3 A schematic of a ring network.

The term *ring* is a misnomer because ring networks don't have an unending cable like a bus network with the two terminators joined together. Instead, the ring refers to the design of the central unit that handles the network's message passing. In a token ring network, the central control unit is called a Media Access Unit, or MAU. The MAU has a ring circuit inside it (for which the network topology is named). The ring inside the MAU serves as the bus for devices to obtain messages.

1.2.1.3 The Hub Network

A hub network uses a main cable much like the bus network, which is called the *backplane*. The hub topology is shown in Figure 1.4. From the backplane, a set of

cables leads to a hub, which is a box containing several ports into which devices are plugged. The cables to a connection point are often called *drops*, because they drop from the backplane to the ports.

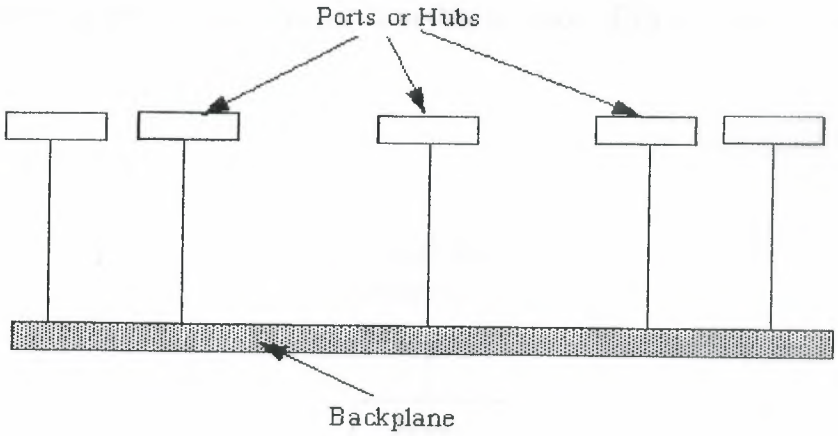

Figure 1.4 A schematic of a hub network.

Hub networks can be very large, using a high-speed fiber optic backplane and slightly slower Ethernet drops to hubs from which a workgroup can be supported. The hub network can also be small, with a couple of hubs supporting a few devices connected together by standard Ethernet cables. The hub network is scaleable (meaning we can start small and expand as we need to), which is part of its attraction.

Hub networks have become popular for large installations, in part because they are easy to set up and maintain. They also can be the least expensive system in many larger installations, which adds to their attraction. The backplane can extend across a considerable distance just like a bus network, whereas the ports, or connection points, are usually grouped in a set placed in a box or panel. There can be many panels or connection boxes attached to the backplane.

1.2.2 Wide Area Networks

LANs can be combined into a large entity called a WAN. WANs are usually composed of LANs joined together by a high-speed link (such as a telephone line or dedicated cable). At the entrance to each LAN, one or more machines act as the link

between the LAN and WAN: these are called gateways. A gateway is the interface between a LAN and a WAN. The same applies for any LAN that accesses the Internet: one machine usually acts as the gateway from the LAN to the Internet (which is really just a very large WAN). LANs can be tied to a WAN through a gateway that handles the passage of data between the LAN and WAN backbone. In a simple layout, a router is used to perform this function. This is shown in Figure 1.5.

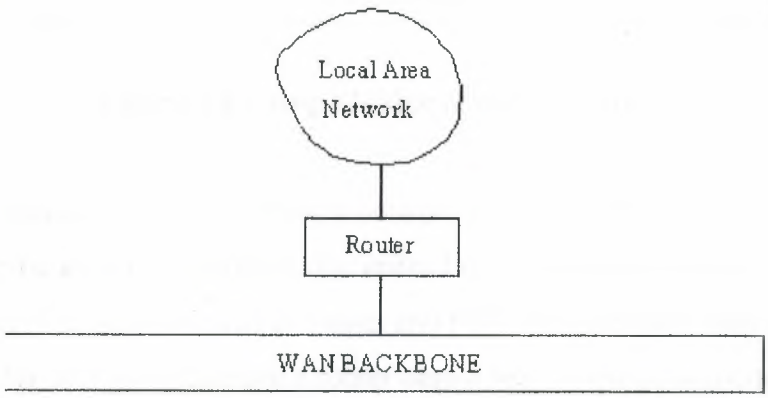

Figure 1.5 A router connects a LAN to the backbone.

Another gateway device, called a bridge, is used to connect LANs using the same network protocol. Bridges are used only when the same network protocol (such as TCP/IP) is on both LANs. The bridge does not care which physical media is used. Bridges can connect twisted-pair LANs to coaxial LANs, for example, or act as an interface to a fiber optic network. As long as the network protocol is the same, the bridge functions properly. If two or more LANs are involved in one organization and there is the possibility of a lot of traffic between them, it is better to connect the two LANs directly with a bridge instead of loading the backbone with the cross-traffic. This is shown in Figure 1.6.

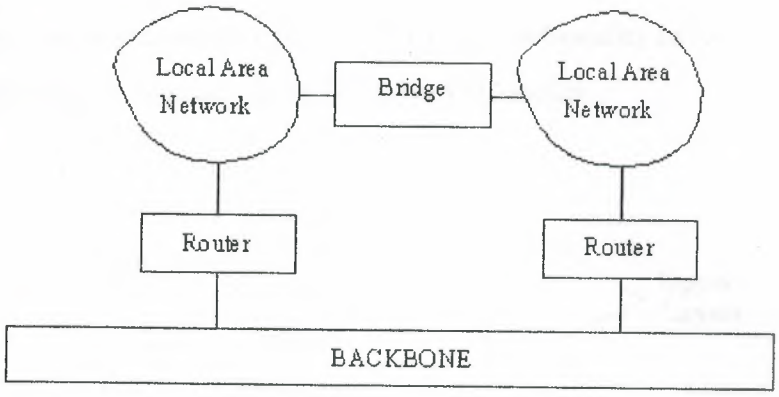

Figure 1.6 Using a bridge to connect two LANs.

In a configuration using bridges between LANs, traffic from one LAN to another can be sent through the bridge instead of onto the backbone, providing better performance. For services such as Telnet and FTP, the speed difference between using a bridge and going through a router onto a heavily used backbone can be significant.

1.3 Layers

Suppose we have to write a program that provides networking functions to every machine on our LAN. Writing a single software package that accomplishes every task required for communications between different computers would be a nightmarish task. Apart from having to cope with the different hardware architectures, simply writing the code for all the applications we desire would result in a program that was far too large to execute or maintain. Dividing all the requirements into similar-purpose groups is a sensible approach, much as a programmer breaks code into logical chunks. With open systems communications, groups are quite obvious. One group deals with the transport of data, another with the packaging of messages, another with end-user applications, and so on. Each group of related tasks is called a *layer*.

Of course, some crossover of functionality is to be expected, and several different approaches to the same division of layers for a network protocol were proposed. One that became adopted as a standard is the Open Systems

Interconnection Reference Model. The OSI Reference Model (OSI-RM) uses seven layers, as shown in Figure 1.7. The TCP/IP architecture is similar but involves only five layers, because it combines some of the OSI functionality in two layers into one. For now, though, we consider the seven-layer OSI model.

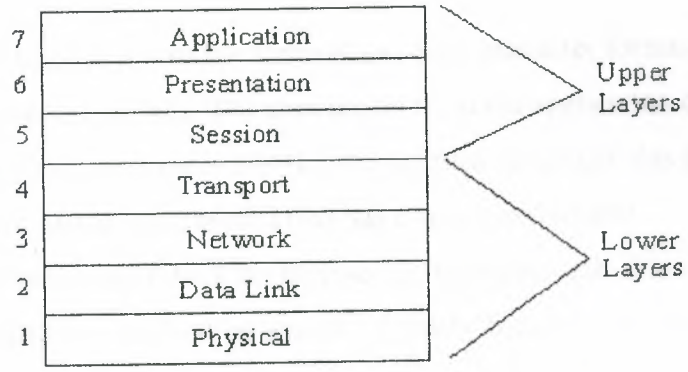

Figure 1.7 The OSI Reference Model showing all seven layers.

The application, presentation, and session layers are all application-oriented in that they are responsible for presenting the application interface to the user. All three are independent of the layers below them and are totally oblivious to the means by which data gets to the application. These three layers are called the upper layers. The lower four layers deal with the transmission of data, covering the packaging, routing, verification, and transmission of each data group. The lower layers don't worry about the type of data they receive or send to the application, but deal simply with the task of sending it. They don't differentiate between the different applications in any way.

1.3.1 The Application Layer

The application layer is the end-user interface to the OSI system. It is where the applications, such as electronic mail, USENET news readers, or database display modules, reside. The application layer's task is to display received information and send the user's new data to the lower layers. In distributed applications, such as client/server systems, the application layer is where the client application resides. It communicates through the lower layers to the server.

1.3.2 The Presentation Layer

The presentation layer's task is to isolate the lower layers from the application's data format. It converts the data from the application into a common format, often called the *canonical representation.* The presentation layer processes machine-dependent data from the application layer into a machine-independent format for the lower layers.

The presentation layer is where file formats and even character formats (ASCII and EBCDIC, for example) are lost. The conversion from the application data format takes place through a "common network programming language" (as it is called in the OSI Reference Model documents) that has a structured format.

The presentation layer does the reverse for incoming data. It is converted from the common format into application-specific formats, based on the type of application the machine has instructions for. If the data comes in without reformatting instructions, the information might not be assembled in the correct manner for the user's application.

1.3.3 The Session Layer

The session layer organizes and synchronizes the exchange of data between application processes. It works with the application layer to provide simple data sets called *synchronization points* that let an application know how the transmission and reception of data are progressing. In simplified terms, the session layer can be thought of as a timing and flow control layer.

The session layer is involved in coordinating communications between different applications, letting each know the status of the other. An error in one application (whether on the same machine or across the country) is handled by the session layer to let the receiving application know that the error has occurred. The session layer can resynchronize applications that are currently connected to each other. This can be necessary when communications are temporarily interrupted, or when an error has occurred that results in loss of data.

1.3.4 The Transport Layer

The transport layer, as its name suggests, is designed to provide the "transparent transfer of data from a source end open system to a destination end open

system," according to the OSI Reference Model. The transport layer establishes, maintains, and terminates communications between two machines.

The transport layer is responsible for ensuring that data sent matches the data received. This verification role is important in ensuring that data is correctly sent, with a resend if an error was detected. The transport layer manages the sending of data, determining its order and its priority.

1.3.5 The Network Layer

The network layer provides the physical routing of the data, determining the path between the machines. The network layer handles all these routing issues, relieving the higher layers from this issue. The network layer examines the network topology to determine the best route to send a message, as well as figuring out relay systems. It is the only network layer that sends a message from source to target machine, managing other chunks of data that pass through the system on their way to another machine.

1.3.6 The Data Link Layer

The data link layer, according to the OSI reference paper, "provides for the control of the physical layer, and detects and possibly corrects errors that can occur." In practicality, the data link layer is responsible for correcting transmission errors induced during transmission (as opposed to errors in the application data itself, which are handled in the transport layer).

The data link layer is usually concerned with signal interference on the physical transmission media, whether through copper wire, fiber optic cable, or microwave. Interference is common, resulting from many sources, including cosmic rays and stray magnetic interference from other sources.

1.3.7 The Physical Layer

The physical layer is the lowest layer of the OSI model and deals with the "mechanical, electrical, functional, and procedural means" required for transmission of data, according to the OSI definition. This is really the wiring or other transmission form. When the OSI model was being developed, a lot of concern dealt with the lower two layers, because they are, in most cases, inseparable. The real world treats the data link layer and the physical layer as one combined layer, but the

formal OSI definition stipulates different purposes for each. (TCP/IP includes the data link and physical layers as one layer, recognizing that the division is more academic than practical.)

1.4 Terminology and Notations

Both OSI and TCP/IP are rooted in formal descriptions, presented as a series of complex documents that define all aspects of the protocols. To define OSI and TCP/IP, several new terms were developed and introduced into use; some (mostly OSI terms) are rather unusual. We might find the term *OSI-speak* used to refer to some of these rather grotesque definitions, much as *legalese* refers to legal terms. To better understand the details of TCP/IP, it is necessary to deal with these terms now. Therefore, all the major terms are covered here.

1.4.1 Packets

To transfer data effectively, many experiments have shown that creating a uniform chunk of data is better than sending characters singly or in widely varying sized groups. Usually these chunks of data have some information ahead of them (the *header*) and sometimes an indicator at the end (the *trailer*). These chunks of data are called *packets* in most synchronous communications systems.

The amount of data in a packet and the composition of the header can change depending on the communications protocol as well as some system limitations, but the concept of a packet always refers to the entire set (including header and trailer). The term *packet* is used often in the computer industry, sometimes when it shouldn't be.

1.4.2 Subsystems

A *subsystem* is the collective of a particular layer across a network. For example, if 10 machines are connected together, each running the seven-layer OSI model, all 10 application layers are the application subsystem, all 10 data link layers are the data link subsystem, and so on. With the OSI Reference Model there are seven subsystems. It is entirely possible that all the individual components in a subsystem will not be active at one time. Using the 10-machine example again, only three might have the data link layer actually active at any moment in time, but the cumulative of all the machines makes up the subsystem.

1.4.3 Entities

A layer can have more than one part to it. For example, the transport layer can have routines that verify checksums as well as routines that handle resending packets that didn't transfer correctly. Not all these routines are active at once, because they might not be required at any moment. The active routines, though, are called entities.

1.4.4 N Notation

The notations N, N+1, N+2, and so on are used to identify a layer and the layers that are related to it. Referring to Figure 1.7, if the transport layer is layer N, the physical layer is N–3 and the presentation layer is N+2. With OSI, N always has a value of 1 through 7 inclusive. One reason this notation was adopted was to enable writers to refer to other layers without having to write out their names every time. It also makes flow charts and diagrams of interactions a little easier to draw. The terms N+1 and N–1 are commonly used in both OSI and TCP for the layers above and below the current layer, respectively.

1.4.5 N-Functions

Each layer performs N-functions. The functions are the different things the layer does. Therefore, the functions of the transport layer are the different tasks that the layer provides. For most purposes, functions and entities mean the same thing.

1.4.6 N-Facilities

This uses the hierarchical layer structure to express the idea that one layer provides a set of facilities to the next higher layer. This is sensible, because the application layer expects the presentation layer to provide a robust, well-defined set of facilities to it. In OSI-speak, the (N+1)-entities assume a defined set of N-facilities from the N-entity.

1.4.7 Services

The entire set of N-facilities provided to the (N+1)-entities is called the N-service. In other words, the service is the entire set of N-functions provided to the next higher layer. Services might seem like functions, but there is a formal difference between the two. The OSI documents go to great lengths to provide detailed descriptions of services, with a "service definition standard" for each layer. This was necessary during the development of the OSI standard so that the different tasks

involved in the communications protocol could be assigned to different layers, and so that the functions of each layer are both well-defined and isolated from other layers.

The service definitions are formally developed from the bottom layer (physical) upward to the top layer. The advantage of this approach is that the design of the N+1 layer can be based on the functions performed in the N layer, avoiding two functions that accomplish the same task in two adjacent layers. An entire set of variations on the service name has been developed to apply these definitions, some of which are in regular use:

An N-service user is a user of a service provided by the N layer to the next higher (N+1) layer.

An N-service provider is the set of N-entities that are involved in providing the N layer service.

An N-service access point (often abbreviated to N-SAP) is where an N-service is provided to an (N+1)-entity by the N-service provider.

N-service data is the packet of data exchanged at an N-SAP.

N-service data units (N-SDUs) are the individual units of data exchanged at an N-SAP (so that N-service data is made up of N-SDUs).

These terms are shown in Figure 1.8. Another common term is *encapsulation,* which is the addition of control information to a packet of data. The control data contains addressing details, checksums for error detection, and protocol control functions.

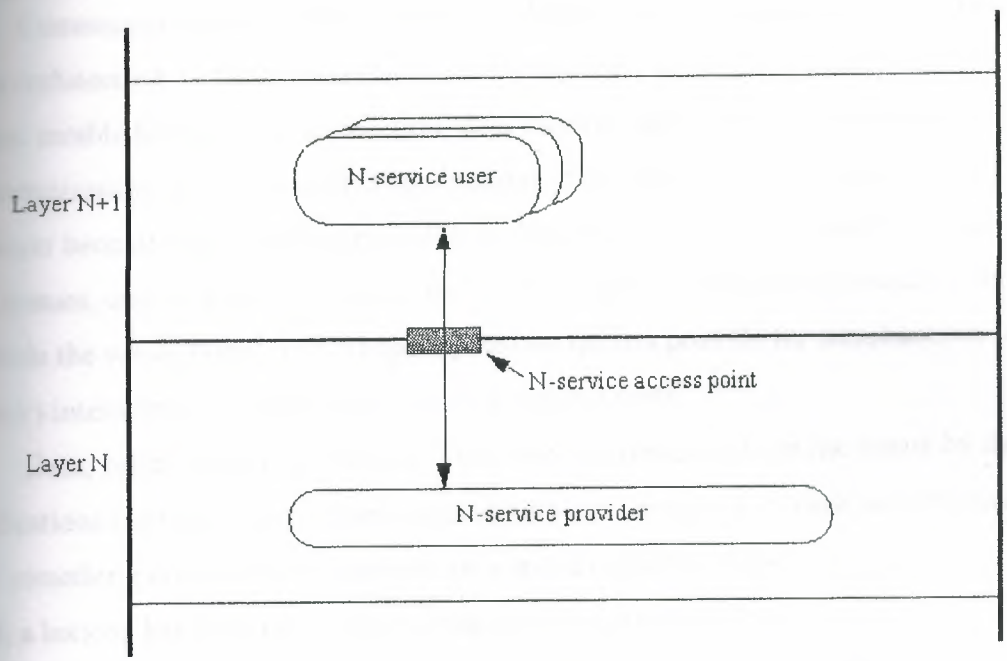

Figure 1.8 Service providers and service users communicate through service access points.

1.4.8 Making Sense of the Jargon

It is important to remember that all these terms are used in a formal description, because a formal language is usually the only method to adequately describe something as complex as a communications protocol. It is possible, though, to fit these terms together so that they make a little more sense when we encounter them. An example should help. The session layer has a set of session functions. It provides a set of session facilities to the layer above it, the presentation layer. The session layer is made up of session entities. The presentation layer is a user of the services provided by the session layer (layer 5). A presentation entity is a user of the services provided by the session layer and is called a presentation service user.

The session service provider is the collection of session entities that are actively involved in providing the presentation layer with the session's services. The point at which the session service is provided to the presentation layer is the session service access point, where the session service data is sent. The individual bits of data in the session service data are called session service data units.

Queues and Connections

Communication between two parties (whether over a telephone, between layers of an architecture, or between applications themselves) takes place in three distinct stages: establishment of the connection, data transfer, and connection termination. Communication between two OSI applications in the same layer is through queues to the layer beneath them. Each application (more properly called a service user) has two queues, one for each direction to the service provider of the layer beneath (which controls the whole layer). In OSI-speak, the two queues provide for simultaneous (or *atomic*) interactions between two N-service action points.

Data, called *service primitives,* is put into and retrieved from the queue by the applications (service users). A service primitive can be a block of data, an indicator that something is required or received, or a status indicator. As with most aspects of OSI, a lexicon has been developed to describe the actions in these queues:

A *request primitive* is when one service submits a service primitive to the queue (through the N-SAP) requesting permission to communicate with another service in the same layer.

An *indication primitive* is what the service provider in the layer beneath the sending application sends to the intended receiving application to let it know that communication is desired.

A *response primitive* is sent by the receiving application to the layer beneath's service provider to acknowledge the granting of communications between the two service users.

A *confirmation primitive* is sent from the service provider to the final application to indicate that both applications on the layer above can now communicate.

An example might help clarify the process. Assume that two applications in the presentation layer want to communicate with each other. They can't do so directly (according to the OSI model), so they must go through the layer below them. These steps are shown in Figure 1.9.

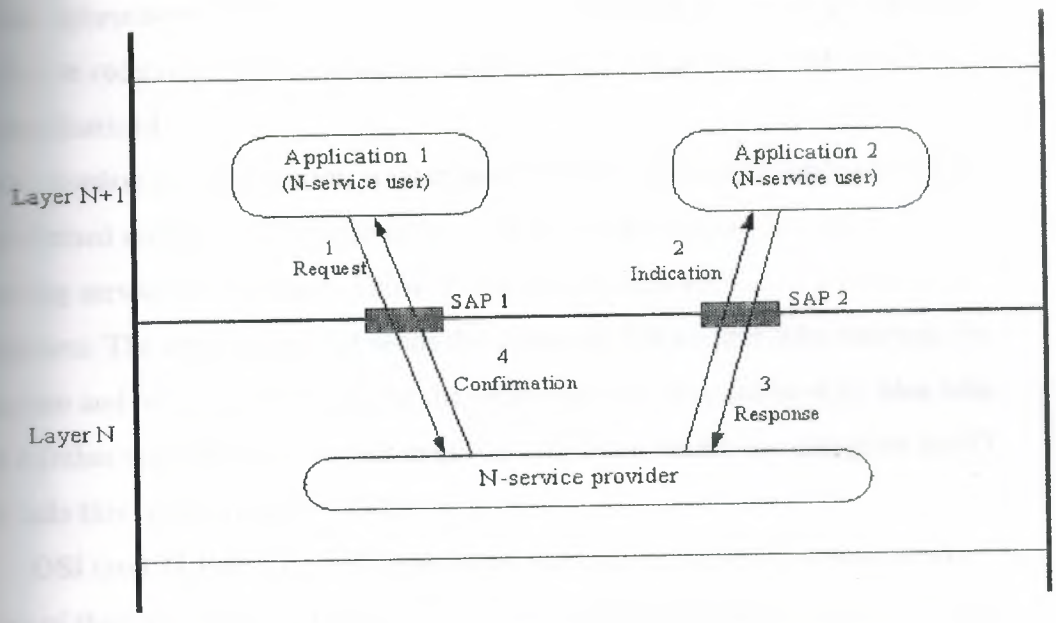

Figure 1.9 Two applications communicate through SAPs using primitives.

The first application sends a request primitive to the service provider of the session layer and waits. The session layer's service provider removes the request primitive from the inbound queue from the first application and sends an indication primitive to the second application's inbound queue.

The second application takes the indication primitive from its queue to the session service provider and decides to accept the request for connection by sending a positive response primitive back through its queue to the session layer. This is received by the session layer service provider, and a confirmation primitive is sent to the first application in the presentation layer. This is a process called *confirmed service* because the applications wait for confirmation that communications are established and ready.

OSI also provides for *unconfirmed service,* in which a request primitive is sent to the service provider, sending the indication primitive to the second application. The response and confirmation primitives are not sent. This is a sort of "get ready, because here it comes whether you want it or not" communication, often referred to as *send and pray.*

When two service users are using confirmed service to communicate, they are considered connected. Two applications are talking to each other, aware of what the

other is doing with the service data. OSI refers to the establishment and maintenance of state *information* between the two, or the fact that each knows when the other is sending or receiving. OSI calls this *connection-oriented* or *connection-mode* communications.

Connectionless communication is when service data is sent independently, as with unconfirmed service. The service data is self-contained, possessing everything a receiving service user needs to know. These service data packets are often called *datagrams.* The application that sends the datagram has no idea who receives the datagram and how it is handled, and the receiving service users have no idea who sent it (other than information that might be contained within the datagram itself). OSI calls this *connectionless-mode.*

OSI (and TCP/IP) use both connected and connectionless systems between layers of their architecture. Each has its benefits and ideal implementations. All these communications are between applications (service users) in each layer, using the layer beneath to communicate. There are many service users, and this process is going on all the time.

1.5 Standards

Standards prevent a situation arising where two seemingly compatible systems really are not. For example, 10 years ago when CP/M was the dominant operating system, the 5.25-inch floppy was used by most systems. But the floppy from a Kaypro II couldn't be read by an Osbourne I because the tracks were laid out in a different manner. A utility program could convert between the two, but that extra step was a major annoyance for machine users. When the IBM PC became the platform of choice, the 5.25-inch format used by the IBM PC was adopted by other companies to ensure disk compatibility. The IBM format became a de facto standard, one adopted because of market pressures and customer demand.

1.5.1 Setting Standards

Creating a standard in today's world is not a simple matter. Several organizations are dedicated to developing the standards in a complete, unambiguous manner. The most important of these is the International Organization for

Standardization, or ISO (often called the International Standardization). The ISO developed the Open Systems Interconnection (OSI) standard.

The goal of ISO is to agree on worldwide standards. Otherwise, incompatibilities could exist that wouldn't allow one country's system to be used in another. (An example of this is with television signals: the US relies on NTSC, whereas Europe uses PAL—systems that are incompatible with each other.) To help define a standard, an abstract approach is usually used. In the case of OSI, the meaning (called the semantics) of the data transferred (the abstract syntax) is first dealt with, and the exact representation of the data in the machine (the concrete syntax) and the means by which it is transferred (transfer syntax) are handled separately. The separation of the abstract lets the data be represented as an entity, without concern for what it really means. To describe systems abstractly, it is necessary to have a language that meets the purpose. Most standards bodies have developed such a system. The most commonly used is ISO's Abstract Syntax Notation One, frequently shortened to ASN.1. It is suited especially for describing open systems networking. Thus, it's not surprising to find it used extensively in the OSI and TCP descriptions. Indeed, ASN.1 was developed concurrently with the OSI standards when it became necessary to describe upper-layer functions.

The primary concept of ASN.1 is that all types of data, regardless of type, size, origin, or purpose, can be represented by an object that is independent of the hardware, operating system software, or application. The ASN.1 system defines the contents of a datagram protocol header—the chunk of information at the beginning of an object that describes the contents to the system.

Part of ASN.1 describes the language used to describe objects and data types (such as a data description language in database terminology). Another part defines the basic encoding rules that deal with moving the data objects between systems. ASN.1 defines data types that are used in the construction of data packets (datagrams). It provides for both structured and unstructured data types, with a list of 28 supported types.

1.5.2 Internet Standards

When the Defense Advanced Research Projects Agency (DARPA) was established in 1980, a group was formed to develop a set of standards for the Internet. The group, called the Internet Configuration Control Board (ICCB) was

reorganized into the Internet Activities Board (IAB) in 1983, whose task was to design, engineer, and manage the Internet. In 1986, the IAB turned over the task of developing the Internet standards to the Internet Engineering Task Force (IETF), and the long-term research was assigned to the Internet Research Task Force (IRTF). The IAB retained final authorization over anything proposed by the two task forces.

The last step in this saga was the formation of the Internet Society in 1992, when the IAB was renamed the Internet Architecture Board. This group is still responsible for existing and future standards, reporting to the board of the Internet Society. After all that, what happened during the shuffling? Almost from the beginning, the Internet was defined as "a loosely organized international collaboration of autonomous, interconnected networks," which supported host-to-host communications "through voluntary adherence to open protocols and procedures" defined in a technical paper called the Internet Standards, RFC 1310,2. That definition is still used today.

The IETF continues to work on refining the standards used for communications over the Internet through a number of working groups, each one dedicated to a specific aspect of the overall Internet protocol suite. There are working groups dedicated to network management, security, user services, routing, and many more things. It is interesting that the IETF's groups are considerably more flexible and efficient than those of, say, the ISO, whose working groups can take years to agree on a standard. In many cases, the IETF's groups can form, create a recommendation, and disband within a year or so. This helps continuously refine the Internet standards to reflect changing hardware and software capabilities.

Creating a new Internet standard (which happened with TCP/IP) follows a well-defined process, shown schematically in Figure 1.10. It begins with a request for comment (RFC). This is usually a document containing a specific proposal, sometimes new and sometimes a modification of an existing standard. RFCs are widely distributed, both on the network itself and to interested parties as printed documents. Important RFCs and instructions for retrieving them are included in the appendixes at the end of this book.

The RFC is usually discussed for a while on the network itself, where anyone can express their opinion, as well as in formal IETF working group meetings. After a suitable amount of revision and continued discussion, an *Internet draft* is created and

distributed. This draft is close to final form, providing a consolidation of all the comments the RFC generated. The next step is usually a *proposed standard,* which remains as such for at least six months. During this time, the Internet Society requires at least two independent and interoperable implementations to be written and tested. Any problems arising from the actual tests can then be addressed. (In practice, it is usual for many implementations to be written and given a thorough testing.)

After that testing and refinement process is completed, a *draft standard* is written, which remains for at least four months, during which time many more implementations are developed and tested. The last step—after many months—is the adoption of the standard, at which point it is implemented by all sites that require it.

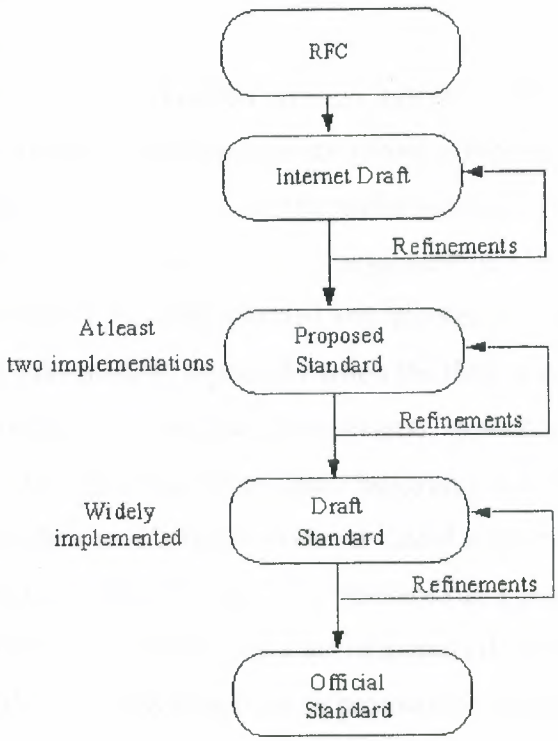

Figure 1.10 The process for adopting a new Internet standard.

1.6 Protocols

Computer protocols define the manner in which communications take place. If one computer is sending information to another and they both follow the protocol properly, the message gets through, regardless of what types of machines they are and what operating systems they run (the basis for open systems). As long as the

machines have software that can manage the protocol, communications are possible. Essentially, a computer protocol is a set of rules that coordinates the exchange of information.

Protocols have developed from very simple processes to elaborate, complex mechanisms that cover all possible problems and transfer conditions. A task such as sending a message from one coast to another can be very complex when we consider the manner in which it moves. A single protocol to cover all aspects of the transfer would be too large, unwieldy, and overly specialized. Therefore, several protocols have been developed, each handling a specific task. Combining several protocols, each with their own dedicated purposes, would be a nightmare if the interactions between the protocols were not clearly defined. The concept of a layered structure was developed to help keep each protocol in its place and to define the manner of interaction between each protocol (essentially, a protocol for communications between protocols!).

The ISO has developed a layered protocol system called OSI. OSI defines a protocol as "a set of rules and formats (semantic and syntactic), which determines the communication behavior of N-entities in the performance of N-functions. *N* represents a layer, and an entity is a service component of a layer. When machines communicate, the rules are formally defined and account for possible interruptions or faults in the flow of information, especially when the flow is connectionless (no formal connection between the two machines exists). In such a system, the ability to properly route and verify each packet of data (datagram) is vitally important. As discussed earlier, the data sent between layers is called a service data unit (SDU), so OSI defines the analogous data between two machines as a protocol data unit (PDU). The flow of information is controlled by a set of actions that define the state machine for the protocol. OSI defines these actions as protocol control information (PCI).

1.6.1 Breaking Data Apart

It is necessary to introduce a few more terms commonly used in OSI and TCP/IP, but luckily they are readily understood because of their real-world connotations. These terms are necessary because data doesn't usually exist in manageable chunks. The data might have to be broken down into smaller sections, or several small sections can be combined into a large section for more efficient transfer. The basic terms are as follows:

Segmentation is the process of breaking an N-service data unit (N-SDU) into several N-protocol data units (N-PDUs).

Reassembly is the process of combining several N-PDUs into an N-SDU (the reverse of segmentation).

Blocking is the combination of several SDUs (which might be from different services) into a larger PDU within the layer in which the SDUs originated.

Unblocking is the breaking up of a PDU into several SDUs in the same layer.

Concatenation is the process of one layer combining several N-PDUs from the next higher layer into one SDU (like blocking except occurring across a layer boundary).

Separation is the reverse of concatenation, so that a layer breaks a single SDU into several PDUs for the next layer higher (like unblocking except across a layer boundary).

These six processes are shown in Figure 1.11.

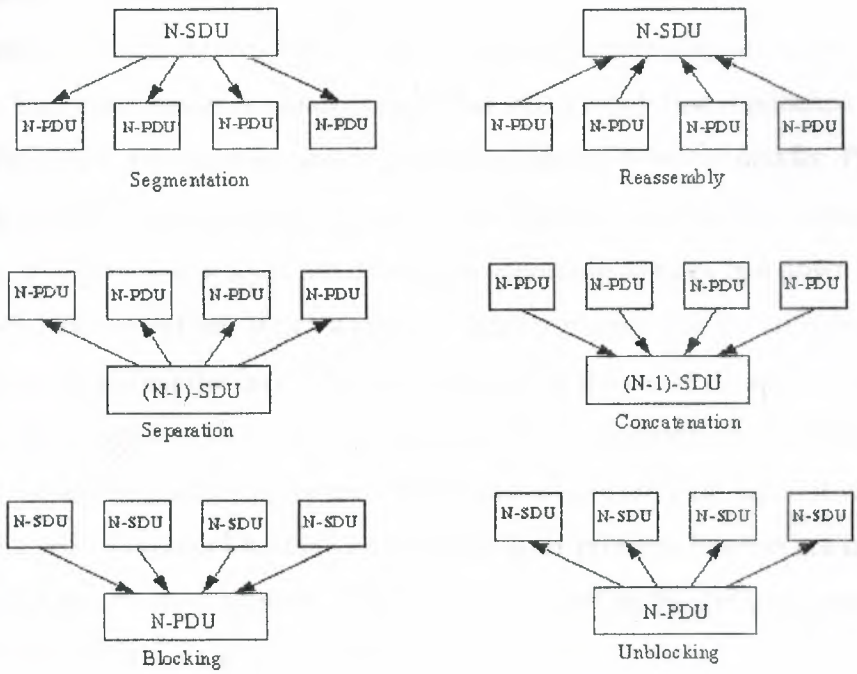

Figure 1.11 Segmentation, reassembly, blocking, unblocking, concatenation, and separation.

Finally, here is one last set of definitions that deal with connections:

Multiplexing is when several connections are supported by a single connection in the next lower layer (so three presentation service connections could be multiplexed into a single session connection).

Demultiplexing is the reverse of multiplexing, in which one connection is split into several connections for the layer above it.

Splitting is when a single connection is supported by several connections in the layer below (so the data link layer might have three connections to support one network layer connection).

Recombining is the reverse of splitting, so that several connections are combined into a single one for the layer above.

Multiplexing and splitting (and their reverses, demultiplexing and recombining) are different in the manner in which the lines are split. With multiplexing, several connections combine into one in the layer below. With splitting, however, one connection can be split into several in the layer below. Each has its importance within TCP and OSI.

1.6.2 Protocol Headers

Protocol control information is information about the datagram to which it is attached. This information is usually assembled into a block that is attached to the front of the data it accompanies and is called a *header* or *protocol header*. Protocol headers are used for transferring information between layers as well as between machines. The protocol headers are developed according to rules laid down in the ISO's ASN.1 document set. When a protocol header is passed to the layer beneath, the datagram including the layer's header is treated as the entire datagram for that receiving layer, which adds its own protocol header to the front. Thus, if a datagram started at the application layer, by the time it reached the physical layer, it would have seven sets of protocol headers on it. These layer protocol headers are used when moving back up the layer structure; they are stripped off as the datagram moves up. An illustration of this is shown in Figure 1.12.

It is easier to think of this process as layers on an onion. The inside is the data that is to be sent. As it passes through each layer of the OSI model, another layer of onion skin is added. When it is finished moving through the layers, several protocol headers are enclosing the data. When the datagram is passed back up the layers

probably on another machine), each layer peels off the protocol header that corresponds to the layer.

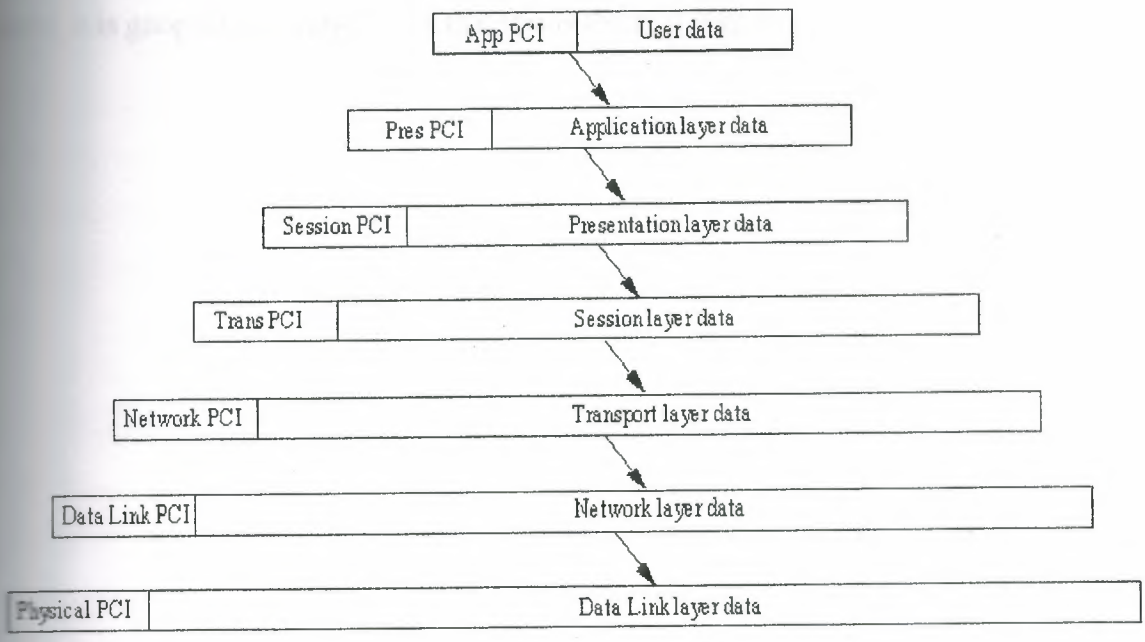

Figure 1.12 Adding each layer's protocol header to user data.

When it reaches the destination layer, only the data is left. This process makes sense, because each layer of the OSI model requires different information from the datagram. By using a dedicated protocol header for each layer of the datagram, it is a relatively simple task to remove the protocol header, decode its instructions, and pass the rest of the message on. The alternative would be to have a single large header that contained all the information, but this would take longer to process.

As usual, OSI has a formal description for all this, which states that the N-user data to be transferred is prepended with N-protocol control information (N-PCI) to form an N-protocol data unit (N-PDU). The N-PDUs are passed across an N-service access point (N-SAP) as one of a set of service parameters comprising an N-service data unit (N-SDU). The service parameters comprising the N-SDU are called N-service user data (N-SUD), which is prepended to the (N–1)PCI to form another (N–1)PDU. For every service in a layer, there is a protocol for it to communicate to the

layer below it (remember that applications communicate through the layer below, not directly). The protocol exchanges for each service are defined by the system, and to a lesser extent by the application developer, who should be following the rules of the system. Protocols and headers might sound a little complex or overly complicated for the task that must be accomplished, but considering the original goals of the OSI model, it is generally acknowledged that this is the best way to go.

CHAPTER TWO

TCP AND UDP

In this chapter we look at the transport layer, where the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) come into play. TCP is one of the most widely used transport layer protocols, expanding from its original implementation on the ARPANET to connecting commercial sites all over the world. In chapter one, "Open Systems, Standards, and Protocols," we looked at the OSI seven-layer model, which bears a striking resemblance to TCP/IP's layered model, so it is not surprising that many of the features of the OSI transport layer were based on TCP.

In theory, a transport layer protocol could be a very simple software routine, but TCP cannot be called simple. Why use a transport layer that is as complex as TCP? The most important reason depends on IP's unreliability. IP does not guarantee delivery of a datagram; it is a connectionless system with no reliability. IP simply handles the routing of datagrams, and if problems occur, IP discards the packet without a second thought (generating an ICMP error message back to the sender in the process). The task of ascertaining the status of the datagrams sent over a network and handling the resending of information if parts have been discarded falls to TCP, which can be thought of as riding shotgun over IP. Most users think of TCP and IP as a tightly knit pair, but TCP can be (and frequently is) used with other protocols without IP. For example, TCP or parts of it are used in the File Transfer Protocol (FTP) and the Simple Mail Transfer Protocol (SMTP), both of which do not use IP.

2.1 What Is TCP?

The Transmission Control Protocol provides a considerable number of services to the IP layer and the upper layers. Most importantly, it provides a connection-oriented protocol to the upper layers that enable an application to be sure that a datagram sent out over the network was received in its entirety. In this role, TCP acts as a message-validation protocol providing reliable communications. If a datagram is corrupted or lost, TCP usually handles the retransmission, rather than the applications in the higher layers. TCP manages the flow of datagrams from the

higher layers to the IP layer, as well as incoming datagrams from the IP layer up to the higher level protocols. TCP has to ensure that priorities and security are properly respected. TCP must be capable of handling the termination of an application above it that was expecting incoming datagrams, as well as failures in the lower layers. TCP also must maintain a state table of all data streams in and out of the TCP layer. The isolation of all these services in a separate layer enables applications to be designed without regard to flow control or message reliability. Without the TCP layer, each application would have to implement the services themselves, which is a waste of resources.

TCP resides in the transport layer, positioned above IP but below the upper layers and their applications, as shown in Figure 2.1. TCP resides only on devices that actually process datagrams, ensuring that the datagram has gone from the source to the target machine. It does not reside on a device that simply routes datagrams, so there is usually no TCP layer in a gateway. This makes sense, because on a gateway the datagram has no need to go higher in the layered model than the IP layer.

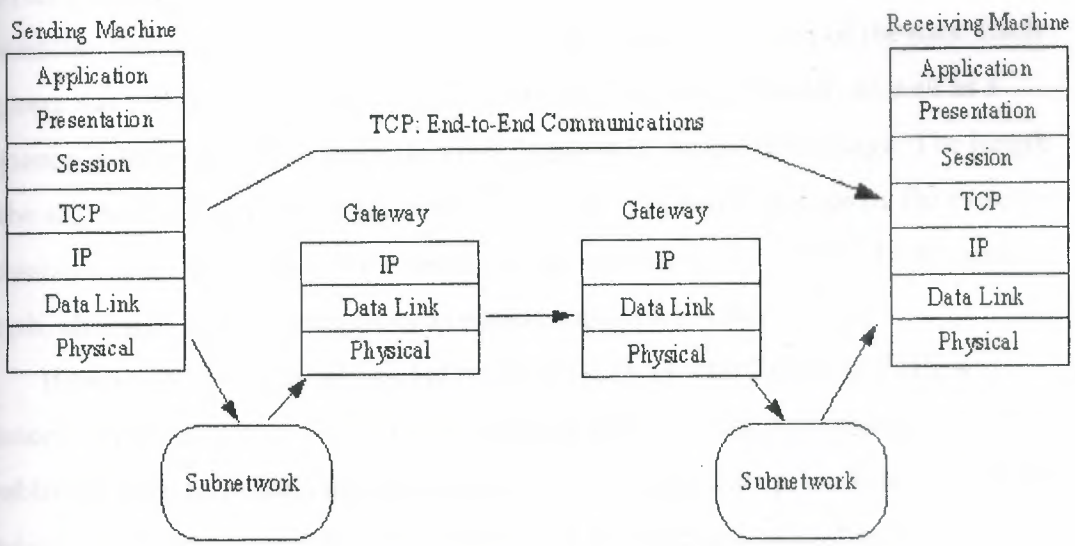

Figure 2.1 TCP provides end-to-end communications.

Because TCP is a connection-oriented protocol responsible for ensuring the transfer of a datagram from the source to destination machine (end-to-end communications), TCP must receive communications messages from the destination machine to acknowledge receipt of the datagram. The term *virtual circuit* is usually used to refer to the communications between the two end machines, most of which are simple acknowledgment messages (either confirmation of receipt or a failure code) and datagram sequence numbers.

2.2 Following a Message

To illustrate the role of TCP, it is instructive to follow a sample message between two machines. The processes are simplified at this stage, to be expanded later. The message originates from an application in an upper layer and is passed to TCP from the next higher layer in the architecture through some protocol (often referred to as an upper-layer protocol, or ULP, to indicate that it resides above TCP). The message is passed as a *stream*—a sequence of individual characters sent asynchronously. This is in contrast to most protocols, which use fixed blocks of data. This can pose some conversion problems with applications that handle only formally constructed blocks of data or insist on fixed-size messages. TCP receives the stream of bytes and assembles them into TCP *segments*, or packets. In the process of assembling the segment, header information is attached at the front of the data. Each segment has a checksum calculated and embedded within the header, as well as a sequence number if there is more than one segment in the entire message. The length of the segment is usually determined by TCP or by a system value set by the system administrator. (The length of TCP segments has nothing to do with the IP datagram length, although there is sometimes a relationship between the two.)

If two-way communications are required (such as with Telnet or FTP), a connection (virtual circuit) between the sending and receiving machines is established prior to passing the segment to IP for routing. This process starts with the sending TCP software issuing a request for a TCP connection with the receiving machine. In the message is a unique number (called a socket number) that identifies the sending machine's connection. The receiving TCP software assigns its own unique socket number and sends it back to the original machine. The two unique

numbers then define the connection between the two machines until the virtual circuit is terminated.

After the virtual circuit is established, TCP sends the segment to the IP software, which then issues the message over the network as a datagram. IP can perform any of the changes to the segment, such as fragmenting it and reassembling it at the destination machine. These steps are completely transparent to the TCP layers, however. After winding its way over the network, the receiving machine's IP passes the received segment up to the recipient machine's TCP layer, where it is processed and passed up to the applications above it using an upper-layer protocol. If the message was more than one TCP segment long (not IP datagrams), the receiving TCP software reassembles the message using the sequence numbers contained in each segment's header. If a segment is missing or corrupt (which can be determined from the checksum), TCP returns a message with the faulty sequence number in the body. The originating TCP software can then resend the bad segment.

If only one segment is used for the entire message, after comparing the segment's checksum with a newly calculated value, the receiving TCP software can generate either a positive acknowledgment (ACK) or a request to resend the segment and route the request back to the sending layer. The receiving machine's TCP implementation can perform a simple flow control to prevent buffer overload. It does this by sending a buffer size called a window value to the sending machine, following which the sender can send only enough bytes to fill the window. After that, the sender must wait for another window value to be received. This provides a handshaking protocol between the two machines, although it slows down the transmission time and slightly increases network traffic. As with most connection-based protocols, timers are an important aspect of TCP. The use of a timer ensures that an undue wait is not involved while waiting for an ACK or an error message. If the timers expire, an incomplete transmission is assumed. Usually an expiring timer before the sending of an acknowledgment message causes a retransmission of the datagram from the originating machine.

Timers can cause some problems with TCP. The specifications for TCP provide for the acknowledgment of only the highest datagram number that has been received without error, but this cannot properly handle fragmentary reception. If a message is composed of several datagrams that arrive out of order, the specification

notes that TCP cannot acknowledge the reception of the message until all the datagrams have been received. So even if all but one datagram in the middle of the sequence have been successfully received, a timer might expire and cause all the datagrams to be resent. With large messages, this can cause an increase in network traffic.

If the receiving TCP software receives duplicate datagrams (as can occur with a retransmission after a timeout or due to a duplicate transmission from IP), the receiving version of TCP discards any duplicate datagrams, without bothering with an error message. After all, the sending system cares only that the message was received—not how many copies were received. TCP does not have a negative acknowledgment (NAK) function; it relies on a timer to indicate lack of acknowledgment. If the timer has expired after sending the datagram without receiving an acknowledgment of receipt, the datagram is assumed to have been lost and is retransmitted. The sending TCP software keeps copies of all unacknowledged datagrams in a buffer until they have been properly acknowledged. When this happens, the retransmission timer is stopped, and the datagram is removed from the buffer. TCP supports a push function from the upper-layer protocols. A push is used when an application wants to send data immediately and confirm that a message passed to TCP has been successfully transmitted. To do this, a push flag is set in the ULP connection, instructing TCP to forward any buffered information from the application to the destination as soon as possible (as opposed to holding it in the buffer until it is ready to transmit it).

2.3 Ports and Sockets

All upper-layer applications that use TCP (or UDP) have a port number that identifies the application. In theory, port numbers can be assigned on individual machines, or however the administrator desires, but some conventions have been adopted to enable better communications between TCP implementations. This enables the port number to identify the type of service that one TCP system is requesting from another. Port numbers can be changed, although this can cause difficulties. Most systems maintain a file of port numbers and their corresponding service.

Typically, port numbers above 255 are reserved for private use of the local machine, but numbers below 255 are used for frequently used processes. A list of frequently used port numbers is published by the Internet Assigned Numbers Authority and is available through an RFC or from many sites that offer Internet summary files for downloading. The commonly used port numbers on this list are shown in Table 2.1. The numbers 0 and 255 are reserved.

Table 2.1. Frequently used TCP port numbers.

Port Number	*Process Name*	*Description*
1	TCPMUX	TCP Port Service Multiplexer
5	RJE	Remote Job Entry
7	ECHO	Echo
9	DISCARD	Discard
11	USERS	Active Users
13	DAYTIME	Daytime
17	Quote	Quotation of the Day
19	CHARGEN	Character generator
20	FTP-DATA	File Transfer Protocol•Data
21	FTP	File Transfer Protocol•Control
23	TELNET	Telnet
25	SMTP	Simple Mail Transfer Protocol
27	NSW-FE	NSW User System Front End
29	MSG-ICP	MSG-ICP
31	MSG-AUTH	MSG Authentication
33	DSP	Display Support Protocol
35		Private Print Servers

37	TIME	Time
38	RLP	Resource Location Protocol
41	GRAPHICS	Graphics
42	NAMESERV	Host Name Server
43	NICNAME	Who Is
49	LOGIN	Login Host Protocol
53	DOMAIN	Domain Name Server
67	BOOTPS	Bootstrap Protocol Server
68	BOOTPC	Bootstrap Protocol Client
69	TFTP	Trivial File Transfer Protocol
79	FINGER	Finger
101	HOSTNAME	NIC Host Name Server
102	ISO-TSAP	ISO TSAP
103	X400	X.400
104	X400SND	X.400 SND
105	CSNET-NS	CSNET Mailbox Name Server
109	POP2	Post Office Protocol v2
110	POP3	Post Office Protocol v3
111	RPC	Sun RPC Portmap
137	NETBIOS-NS	NETBIOS Name Service
138	NETBIOS-DG	NETBIOS Datagram Service
139	NETBIOS-SS	NETBIOS Session Service
146	ISO-TP0	ISO TP0
147	ISO-IP	ISO IP

150	SQL-NET	SQL NET
153	SGMP	SGMP
156	SQLSRV	SQL Service
160	SGMP-TRAPS	SGMP TRAPS
161	SNMP	SNMP
162	SNMPTRAP	SNMPTRAP
163	CMIP-MANAGE	CMIP/TCP Manager
164	CMIP-AGENT	CMIP/TCP Agent
165	XNS-Courier	Xerox
179	BGP	Border Gateway Protocol

Each communication circuit into and out of the TCP layer is uniquely identified by a combination of two numbers, which together are called a socket. The socket is composed of the IP address of the machine and the port number used by the TCP software. Both the sending and receiving machines have sockets. Because the IP address is unique across the internetwork, and the port numbers are unique to the individual machine, the socket numbers are also unique across the entire internetwork. This enables a process to talk to another process across the network, based entirely on the socket number. The last section examined the process of establishing a message. During the process, the sending TCP requests a connection with the receiving TCP, using the unique socket numbers. This process is shown in Figure 2.2. If the sending TCP wants to establish a Telnet session from its port number 350, the socket number would be composed of the source machine's IP address and the port number (350), and the message would have a destination port number of 23 (Telnet's port number). The receiving TCP has a source port of 23 (Telnet) and a destination port of 350 (the sending machine's port).

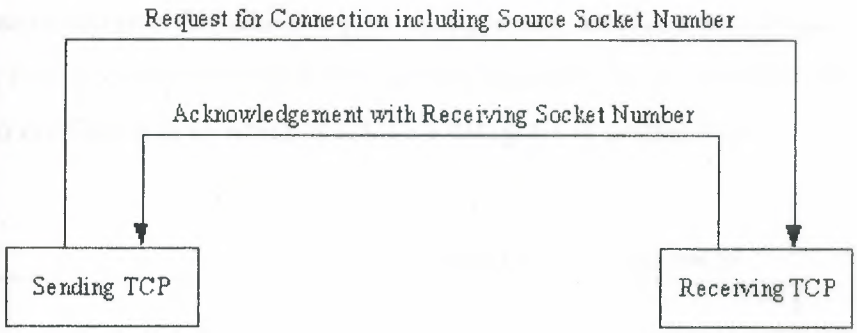

Figure 2.2 Setting up a virtual circuit with socket numbers.

The sending and receiving machines maintain a port table, which lists all active port numbers. The two machines involved have reversed entries for each session between the two. This is called binding and is shown in Figure 2.3. The source and destination numbers are simply reversed for each connection in the port table. Of course, the IP addresses, and hence the socket numbers, are different.

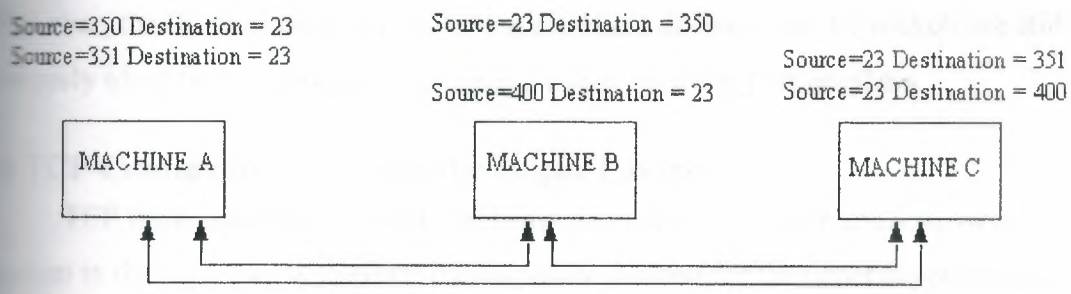

Figure 2.3 Binding entries in port tables.

If the sending machine is requesting more than one connection, the source port numbers are different, even though the destination port numbers might be the same. For example, if the sending machine were trying to establish three Telnet sessions simultaneously, the source machine port numbers might be 350, 351, and 352, and

the destination port numbers would all be 23. It is possible for more than one machine to share the same destination socket—a process called multiplexing. In Figure 2.4, three machines are establishing Telnet sessions with a destination. They all use destination port 23, which is port multiplexing. Because the datagrams emerging from the port have the full socket information (with unique IP addresses), there is no confusion as to which machine a datagram is destined for.

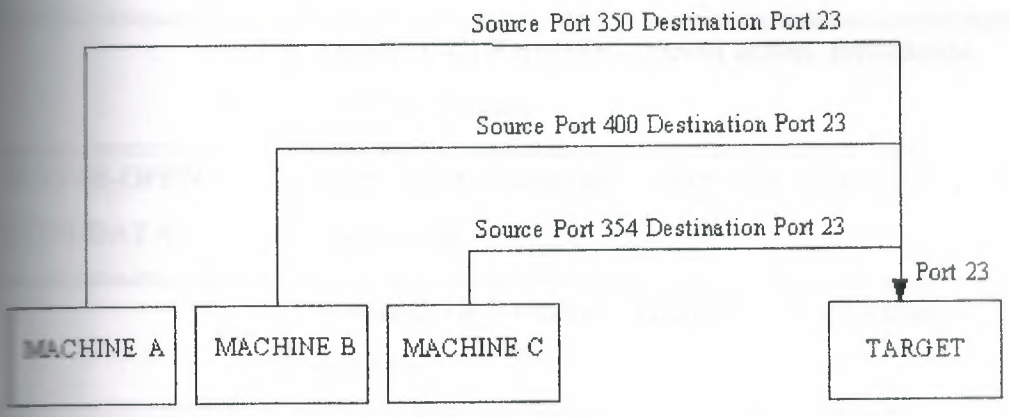

Figure 2.4 Multiplexing one destination port.

When multiple sockets are established, it is conceivable that more than one machine might send a connection request with the same source and destination ports. However, the IP addresses for the two machines are different, so the sockets are still uniquely identified despite identical source and destination port numbers.

2.4 TCP Communications with the Upper Layers

TCP must communicate with applications in the upper layer and a network system in the layer below. Several messages are defined for the upper-layer protocol to TCP communications, but there is no defined method for TCP to talk to lower layers (usually, but not necessarily, IP). TCP expects the layer beneath it to define the communication method. It is usually assumed that TCP and the transport layer communicate asynchronously. The TCP to upper-layer protocol (ULP) communication method is well-defined, consisting of a set of service request primitives. The primitives involved in ULP to TCP communications are shown in Table 2.2.

Table 2.2. ULP-TCP service primitives.

Command	*Parameters Expected*
ULP to TCP Service Request Primitives	
ABORT	Local connection name
ACTIVE-OPEN	Local port, remote socket
	Optional: ULP timeout, timeout action, precedence, security, options
ACTIVE-OPEN-WITH-DATA	Source port, destination socket, data, data length, push flag, urgent flag
	Optional: ULP timeout, timeout action, precedence, security
ALLOCATE	Local connection name, data length
CLOSE	Local connection name
FULL-PASSIVE-OPEN	Local port, destination socket
	Optional: ULP timeout, timeout action, precedence, security, options
RECEIVE	Local connection name, buffer address, byte count, push flag, urgent flag
SEND	Local connection name, buffer address, data length, push flag, urgent flag
	Optional: ULP timeout, timeout action
STATUS	Local connection name
UNSPECIFIED-PASSIVE-OPEN	Local port

	Optional: ULP timeout, timeout action, precedence, security, options
TCP to ULP Service Request Primitives	
CLOSING	Local connection name
DELIVER	Local connection name, buffer address, data length, urgent flag
ERROR	Local connection name, error description
OPEN-FAILURE	Local connection name
OPEN-ID	Local connection name, remote socket, destination address
OPEN-SUCCESS	Local connection name
STATUS RESPONSE	Local connection name, source port, source address, remote socket, connection state, receive window, send window, amount waiting ACK, amount waiting receipt, urgent mode, precedence, security, timeout, timeout action
TERMINATE	Local connection name, description

2.5 Passive and Active Ports

TCP enables two methods to establish a connection: active and passive. An active connection establishment happens when TCP issues a request for the connection, based on an instruction from an upper-level protocol that provides the socket number. A passive approach takes place when the upper-level protocol instructs TCP to wait for the arrival of connection requests from a remote system (usually from an active open instruction). When TCP receives the request, it assigns a port number. This enables a connection to proceed rapidly, without waiting for the active process. There are two passive open primitives. A specified passive open creates a connection when the precedence level and security level are acceptable. An unspecified passive open opens the port to any request. The latter is used by servers that are waiting for clients of an unknown type to connect to them.

TCP has strict rules about the use of passive and active connection processes. Usually a passive open is performed on one machine, while an active open is performed on the other, with specific information about the socket number, precedence (priority), and security levels. Although most TCP connections are established by an active request to a passive port, it is possible to open a connection without a passive port waiting. In this case, the TCP that sends a request for a connection includes both the local socket number and the remote socket number. If the receiving TCP is configured to enable the request (based on the precedence and security settings, as well as application-based criteria), the connection can be opened. This process is looked at again in the section titled "TCP and Connections."

2.6 TCP Timers

TCP uses several timers to ensure that excessive delays are not encountered during communications. Several of these timers are elegant, handling problems that are not immediately obvious at first analysis. The timers used by TCP are examined in the following sections, which reveal their roles in ensuring that data is properly sent from one connection to another.

2.6.1 The Retransmission Timer

The retransmission timer manages retransmission timeouts (RTOs), which occur when a preset interval between the sending of a datagram and the returning acknowledgment is exceeded. The value of the timeout tends to vary, depending on the network type, to compensate for speed differences. If the timer expires, the datagram is retransmitted with an adjusted RTO, which is usually increased exponentially to a maximum preset limit. If the maximum limit is exceeded, connection failure is assumed, and error messages are passed back to the upper-layer application. Values for the timeout are determined by measuring the average time that data takes to be transmitted to another machine and the acknowledgment received back, which is called the round-trip time, or RTT. From experiments, these RTTs are averaged by a formula that develops an expected value, called the smoothed round-trip time, or SRTT. This value is then increased to account for unforeseen delays.

2.6.2 The Quiet Timer

After a TCP connection is closed, it is possible for datagrams that are still making their way through the network to attempt to access the closed port. The quiet timer is intended to prevent the just-closed port from reopening again quickly and receiving these last datagrams. The quiet timer is usually set to twice the maximum segment lifetime (the same value as the Time to Live field in an IP header), ensuring that all segments still heading for the port have been discarded. Typically, this can result in a port being unavailable for up to 30 seconds, prompting error messages when other applications attempt to access the port during this interval.

2.6.3 The Persistence Timer

The persistence timer handles a fairly rare occurrence. It is conceivable that a receive window might have a value of 0, causing the sending machine to pause transmission. The message to restart sending might be lost, causing an infinite delay. The persistence timer waits a preset time and then sends a one-byte segment at predetermined intervals to ensure that the receiving machine is still clogged. The receiving machine resends the zero window-size message after receiving one of these status segments, if it is still backlogged. If the window is open, a message giving the new value is returned, and communications are resumed.

2.6.4 The Keep-Alive Timer and the Idle Timer

Both the keep-alive timer and the idle timer were added to the TCP specifications after their original definition. The keep-alive timer sends an empty packet at regular intervals to ensure that the connection to the other machine is still active. If no response has been received after sending the message by the time the idle timer has expired, the connection is assumed to be broken. The keep-alive timer value is usually set by an application, with values ranging from 5 to 45 seconds. The idle timer is usually set to 360 seconds.

2.7 Transmission Control Blocks and Flow Control

TCP has to keep track of a lot of information about each connection. It does this through a Transmission Control Block (TCB), which contains information about the local and remote socket numbers, the send and receive buffers, security and

values, and the current segment in the queue. The TCB also manages send and receive sequence numbers. The TCB uses several variables to keep track of the send and receive status and to control the flow of information. These variables are shown in Table 2.3.

Table 2.3. TCP send and receive variables.

Variable Name	*Description*
Send Variables	
SND.UNA	Send Unacknowledged
SND.NXT	Send Next
SND.WND	Send Window
SND.UP	Sequence number of last urgent set
SND.WL1	Sequence number for last window update
SND.WL2	Acknowledgment number for last window update
SND.PUSH	Sequence number of last pushed set
ISS	Initial send sequence number
Receive Variables	
RCV.NXT	Sequence number of next received set
RCV.WND	Number of sets that can be received
RCV.UP	Sequence number of last urgent data
RCV.IRS	Initial receive sequence number

Using these variables, TCP controls the flow of information between two sockets. A sample connection session helps illustrate the use of the variables. It begins with Machine A wanting to send five blocks of data to Machine B. If the window limit is seven blocks, a maximum of seven blocks can be sent without acknowledgment. The SND.UNA variable on Machine A indicates how many blocks

have been sent but are unacknowledged (5), and the SND.NXT variable has the value of the next block in the sequence (6). The value of the SND.WND variable is 2 (seven blocks possible, minus five sent), so only two more blocks could be sent without overloading the window. Machine B returns a message with the number of blocks received, and the window limit is adjusted accordingly. The passage of messages back and forth can become quite complex as the sending machine forwards blocks unacknowledged up to the window limit, waiting for acknowledgment of earlier blocks that have been removed from the incoming cue, and then sending more blocks to fill the window again. The tracking of the blocks becomes a matter of bookkeeping, but with large window limits and traffic across internetworks that sometimes cause blocks to go astray, the process is, in many ways, remarkable.

2.8 TCP Protocol Data Units

As mentioned earlier, TCP must communicate with IP in the layer below (using an IP-defined method) and applications in the upper layer (using the TCP-ULP primitives). TCP also must communicate with other TCP implementations across networks. To do this, it uses Protocol Data Units (PDUs), which are called segments in TCP parlance.

The layout of the TCP PDU (commonly called the header) is shown in Figure 2.5.

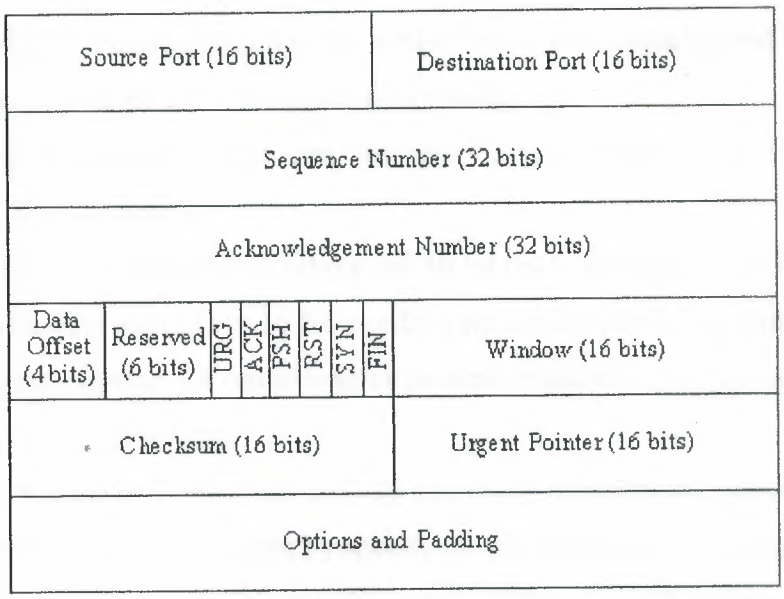

Figure 2.5 The TCP Protocol Data Unit.

The different fields are as follows:

- Source port: A 16-bit field that identifies the local TCP user (usually an upper-layer application program).
- Destination port: A 16-bit field that identifies the remote machine's TCP user.
- Sequence number: A number indicating the current block's position in the overall message. This number is also used between two TCP implementations to provide the initial send sequence (ISS) number.
- Acknowledgment number: A number that indicates the next sequence number expected. In a backhanded manner, this also shows the sequence number of the last data received; it shows the last sequence number received plus 1.
- Data offset: The number of 32-bit words that are in the TCP header. This field is used to identify the start of the data field.
- Reserved: A 6-bit field reserved for future use. The six bits must be set to 0.
- Urg flag: If on (a value of 1), indicates that the urgent pointer field is significant.
- Ack flag: If on, indicates that the Acknowledgment field is significant.
- Psh flag: If on, indicates that the push function is to be performed.
- Rst flag: If on, indicates that the connection is to be reset.
- Syn flag: If on, indicates that the sequence numbers are to be synchronized. This flag is used when a connection is being established.
- Fin flag: If on, indicates that the sender has no more data to send. This is the equivalent of an end-of-transmission marker.
- Window: A number indicating how many blocks of data the receiving machine can accept.
- Checksum: Calculated by taking the 16-bit one's complement of the one's complement sum of the 16-bit words in the header (including pseudo-header) and text together. (A rather lengthy process required to fit the checksum properly into the header.)
- Urgent pointer: Used if the urg flag was set; it indicates the portion of the data message that is urgent by specifying the offset from the sequence number in the header. No specific action is taken by TCP with respect to urgent data; the action is determined by the application.

- Options: Similar to the IP header option field, this is used for specifying TCP options. Each option consists of an option number (one byte), the number of bytes in the option, and the option values. Only three options are currently defined for TCP:

  0 End of option list

  1 No operation

  2 Maximum segment size
- Padding: Filled to ensure that the header is a 32-bit multiple.

Following the PDU or header is the data. The Options field has one useful function: to specify the maximum buffer size a receiving TCP implementation can accommodate. Because TCP uses variable-length data areas, it is possible for a sending machine to create a segment that is longer than the receiving software can handle. The Checksum field calculates the checksum based on the entire segment size, including a 96-bit pseudoheader that is prefixed to the TCP header during the calculation. The pseudoheader contains the source address, destination address, protocol identifier, and segment length. These are the parameters that are passed to IP when a send instruction is passed, and also the ones read by IP when delivery is attempted.

2.9 TCP and Connections

TCP has many rules imposed on how it communicates. These rules and the processes that TCP follows to establish a connection, transfer data, and terminate a connection are usually presented in state diagrams. (Because TCP is a state-driven protocol, its actions depend on the state of a flag or similar construct.) Avoiding overly complex state diagrams is difficult, so flow diagrams can be used as a useful method for understanding TCP.

2.9.1 Establishing a Connection

A connection can be established between two machines only if a connection between the two sockets does not exist, both machines agree to the connection, and

both machines have adequate TCP resources to service the connection. If any of these conditions are not met, the connection cannot be made. The acceptance of connections can be triggered by an application or a system administration routine.

When a connection is established, it is given certain properties that are valid until the connection is closed. Typically, these are a precedence value and a security value. These settings are agreed upon by the two applications when the connection is in the process of being established. In most cases, a connection is expected by two applications, so they issue either active or passive open requests. Figure 2.6 shows a flow diagram for a TCP open. The process begins with Machine A's TCP receiving a request for a connection from its ULP, to which it sends an active open primitive to Machine B. (Refer back to Table 4.2 for the TCP primitives.) The segment that is constructed has the SYN flag set on (set to 1) and has a sequence number assigned. The diagram shows this with the notation "SYN SEQ 50," indicating that the SYN flag is on and the sequence number (Initial Send Sequence number or ISS) is 50. (Any number could have been chosen.)

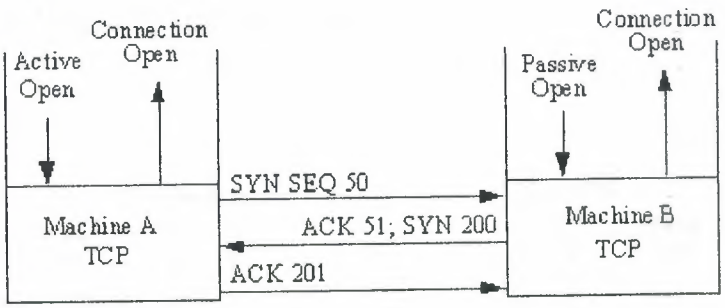

Figure 2.6 Establishing a connection.

The application on Machine B has issued a passive open instruction to its TCP. When the SYN SEQ 50 segment is received, Machine B's TCP sends an acknowledgment back to Machine A with the sequence number of 51. Machine B also sets an ISS number of its own. The diagram shows this message as "ACK 51; SYN 200," indicating that the message is an acknowledgment with sequence number

it has the SYN flag set, and it has an ISS of 200. Upon receipt, Machine A sends back its own acknowledgment message with the sequence number set to 201. This is "ACK 201" in the diagram. Then, having opened and acknowledged the connection, Machine A and Machine B both send connection open messages through the ULP to the requesting applications.

It is not necessary for the remote machine to have a passive open instruction, as mentioned earlier. In this case, the sending machine provides both the sending and receiving socket numbers, as well as precedence, security, and timeout values. It is common for two applications to request an active open at the same time. This is resolved quite easily, although it does involve a little more network traffic.

2.9.2 Data Transfer

Transferring information is straightforward, as shown in Figure 4.7. For each block of data received by Machine A's TCP from the ULP, TCP encapsulates it and sends it to Machine B with an increasing sequence number. After Machine B receives the message, it acknowledges it with a segment acknowledgment that increments the next sequence number (and hence indicates that it has received everything up to that sequence number). Figure 2.7 shows the transfer of two segments of information—one each way.

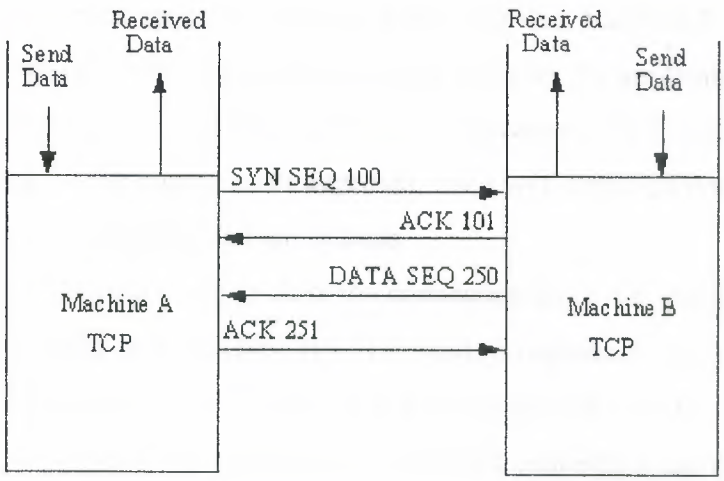

Figure 2.7 Data transfers.

The TCP data transport service actually embodies six subservices:

- Full duplex: Enables both ends of a connection to transmit at any time, even simultaneously.
- Timeliness: The use of timers ensures that data is transmitted within a reasonable amount of time.
- Ordered: Data sent from one application is received in the same order at the other end. This occurs despite the fact that the datagrams might be received out of order through IP, because TCP reassembles the message in the correct order before passing it up to the higher layers.
- Labeled: All connections have an agreed-upon precedence and security value.
- Controlled flow: TCP can regulate the flow of information through the use of buffers and window limits.
- Error correction: Checksums ensure that data is free of errors (within the checksum algorithm's limits).

2.9.3 Closing Connections

To close a connection, one of the TCPs receives a close primitive from the ULP and issues a message with the FIN flag set on. This is shown in Figure 2.8. In the figure, Machine A's TCP sends the request to close the connection to Machine B with the next sequence number. Machine B then sends back an acknowledgment of the request and its next sequence number. Following this, Machine B sends the close message through its ULP to the application and waits for the application to acknowledge the closure. This step is not strictly necessary; TCP can close the connection without the application's approval, but a well-behaved system would inform the application of the change in state.

After receiving approval to close the connection from the application (or after the request has timed out), Machine B's TCP sends a segment back to Machine A with the FIN flag set. Finally, Machine A acknowledges the closure, and the connection is terminated. An abrupt termination of a connection can occur when one side shuts down the socket. This can be done without any notice to the other machine and without regard to any information in transit between the two. Aside from sudden shutdowns caused by malfunctions or power outages, abrupt termination can be

initiated by a user, an application, or a system monitoring routine that judges the connection worthy of termination. The other end of the connection might not realize that an abrupt termination has occurred until it attempts to send a message and the timer expires.

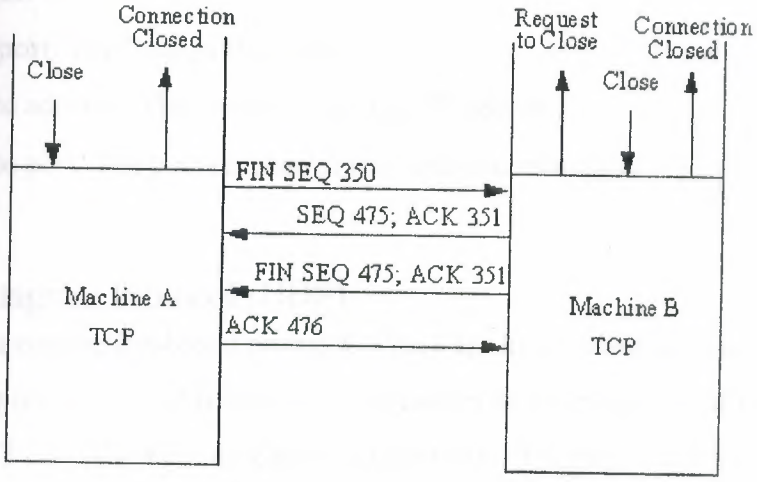

Figure 2.8 Closing a connection.

To keep track of all the connections, TCP uses a connection table. Each existing connection has an entry in the table that shows information about the end-to-end connection. The layout of the TCP connection table is shown in Figure 2.9.

	STATE	LOCAL ADDRESS	LOCAL PORT	REMOTE ADDRESS	REMOTE PORT
Connection 1					
Connection 2					
Connection 3					
Connection n					

Figure 2.9 The TCP connection table.

The meaning of each column is as follows:

- State: The state of the connection (closed, closing, listening, waiting, and so on).
- Local address: The IP address for the connection. When in a listening state, this is set to 0.0.0.0.
- Local port: The local port number.
- Remote address: The remote machine's IP address.
- Remote port: The port number of the remote connection.

2.10 User Datagram Protocol (UDP)

TCP is a connection-based protocol. There are times when a connectionless protocol is required, so UDP is used. UDP is used with both the Trivial File Transfer Protocol (TFTP) and the Remote Call Procedure (RCP). Connectionless communications don't provide reliability, meaning there is no indication to the sending device that a message has been received correctly. Connectionless protocols also do not offer error-recovery capabilities—which must be either ignored or provided in the higher or lower layers. UDP is much simpler than TCP. It interfaces with IP (or other protocols) without the bother of flow control or error-recovery mechanisms, acting simply as a sender and receiver of datagrams. The UDP message header is much simpler than TCP's. It is shown in Figure 2.10. Padding can be added to the datagram to ensure that the message is a multiple of 16 bits.

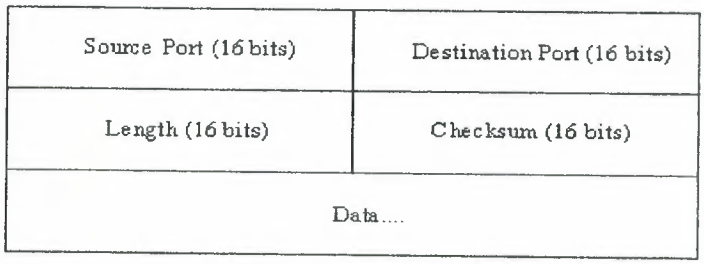

Figure 2.10 The UDP header.

The fields are as follows:

- Source port: An optional field with the port number. If a port number is not specified, the field is set to 0.
- Destination port: The port on the destination machine.
- Length: The length of the datagram, including header and data.
- Checksum: A 16-bit one's complement of the one's complement sum of the datagram, including a pseudoheader similar to that of TCP.

The UDP checksum field is optional, but if it isn't used, no checksum is applied to the data segment because IP's checksum applies only to the IP header. If the checksum is not used, the field should be set to 0.

CHAPTER THREE

TCP/UDP AND NETWORKS

TCP/UDP and Other Protocols

TCP/IP is not often found as a sole protocol. It is usually one of several protocols used in any given network. Therefore, the interactions between TCP/UDP (and its associated protocols) and the other protocols that might be working with it must be understood. It is easiest to begin looking at this subject from a local area network point of view and then expand that view to cover internetworks. The layers of a TCP/IP protocol, as well as most other OSI-model protocols, are designed to be independent of each other, enabling mixing of protocols. When a message is to be sent over the network to a remote machine, each protocol layer builds on the packet of information sent from the layer above, adding its own header and then passing the packet to the next lower layer. After being received over the network (packaged in whatever network format is required), the receiving machine passes the packet back up the layers, removing the header information one layer at a time.

Replacing any layer in the protocol stack requires that the new protocols can internetwork with the other layers, as well as perform all the required functions of that layer (for example, duplicating the services of the replaced protocol). To examine the internetworking of the layers and the substitution or addition of others, a simple installation can be used as a starting point. Figure 3.1 shows a simple layered architecture using TCP and IP with the Ethernet network. Figure 3.1 also shows the assembly of Ethernet packets as they pass from layer to layer.

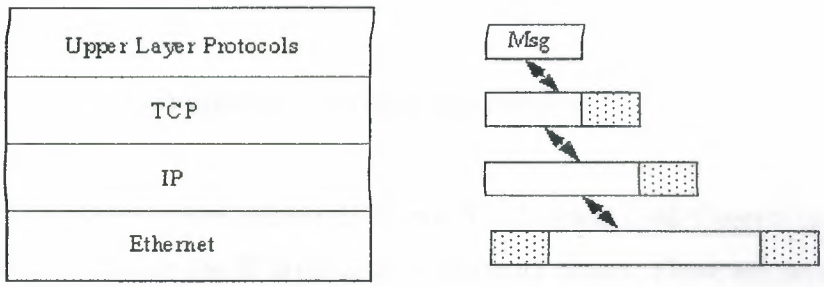

Figure 3.1 A simple layered architecture.

The process begins with a message of some form from an Upper Layer Protocol (ULP) which itself is passing a message from an application. As the message is passed to TCP, it adds its own header information and passes to the IP layer, which does the same. When the IP message is passed to the Ethernet layer, Ethernet adds its own information at the front and back of the message and sends the message out over the network. Although this simple model might seem ideal, in practice it has a few problems. Most importantly, it requires IP to interface directly with the Ethernet layer. This interface is not a clean one; it has many connections that break from the ideal layered architecture.

3.1.1 LAN Layers

To expand on the layered system requires a better understanding of the interfaces to the network layer in a LAN. Figure 3.2 shows an expanded layer architecture for a LAN. This type of architecture applies for collision sense multiple access (CSMA) and collision detect (CD) networks such as Ethernet.

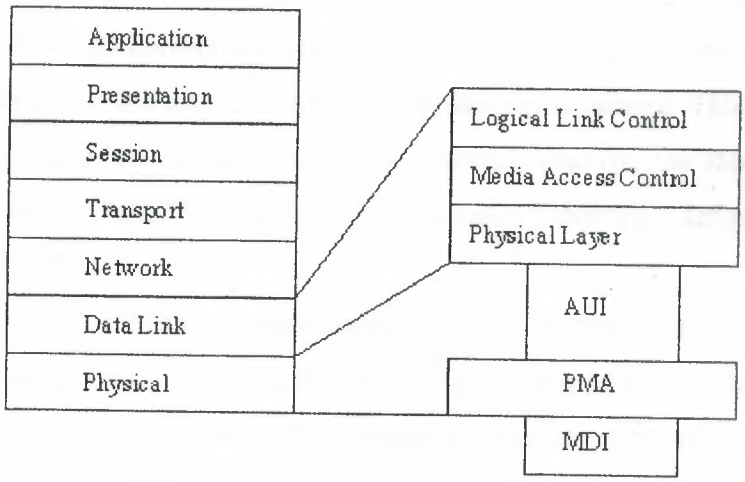

Figure 3.2 Network architecture.

The LAN involves some additional layers. The Logical Link Control (LLC) layer is an interface between the IP layer and the network layers. There are several kinds of LLC configurations, but it is sufficient at this point to know its basic role as a buffer between the network and IP layers either as a simple system for a connectionless service or as an elaborate system for a connection-based service. LLC is usually used with the

High-Level Data Link Control (HDLC) link standard. For connectionless service, this uses an *unnumbered information* (UI) message frame, whereas connection-based services can use the *asynchronous balanced mode* (ABM) message frame, both supported by HDLC. The configuration of LLC with respect to TCP/IP is important.

MAC is responsible for managing traffic on the network, such as collision detection and transmission times. It also handles timers and retransmission functions. MAC is independent of the network medium but is dependent on the protocol used on the network. The physical layer in the Network architecture is composed of several services. The Attachment Unit Interface (AUI) provides an attachment between the machine's physical layer and the network medium. Typically, the AUI is where the network ports or jacks are located.

The Medium Attachment Unit (MAU) is composed of two parts: the Physical Medium Attachment (PMA) and the Medium Dependent Interface (MDI), both of which can be considered as separate parts as shown in the figure. The MAU is responsible for managing the connection of the machine to the LAN medium itself, as well as providing basic data integrity checking and network medium monitoring. The MAU has functions that check the signal quality from the network and test routines for verifying the network's correct operation. When these layers are added to the layered architecture for a protocol stack, the IP-Ethernet layer is separated. This is shown in Figure 3.3. This type of configuration is more common than the one shown in Figure 3.1 and is usually called the IP/802 configuration (because Ethernet is defined by the IEEE 802 specification).

The IP/802 LAN can be connectionless using a simple form of LLC called LLC Type 1, which supports unnumbered information (UI). The LLC and MAC layers help separate IP from the physical layer. More headers are added to the message packet, but these have useful information. The LLC header has both source and destination service access points (SAP) in it to identify the layers above. UDP is frequently used instead of TCP in this type of network. UDP is not as complex as TCP, so the entire network's complexity is reduced. However, UDP has no message integrity functionality built in, so a different form of LLC (called LLC Type 2) is used that implements these functions. LLC Type 2 provides the data integrity functionality that TCP usually provides, such as sequencing, transfer window management, and flow control. The disadvantage is that these functions are now below the IP layer, instead of above it. In case of fatal problems

with the LLC layer, this can result in problems that must be dealt with in the application layer itself.

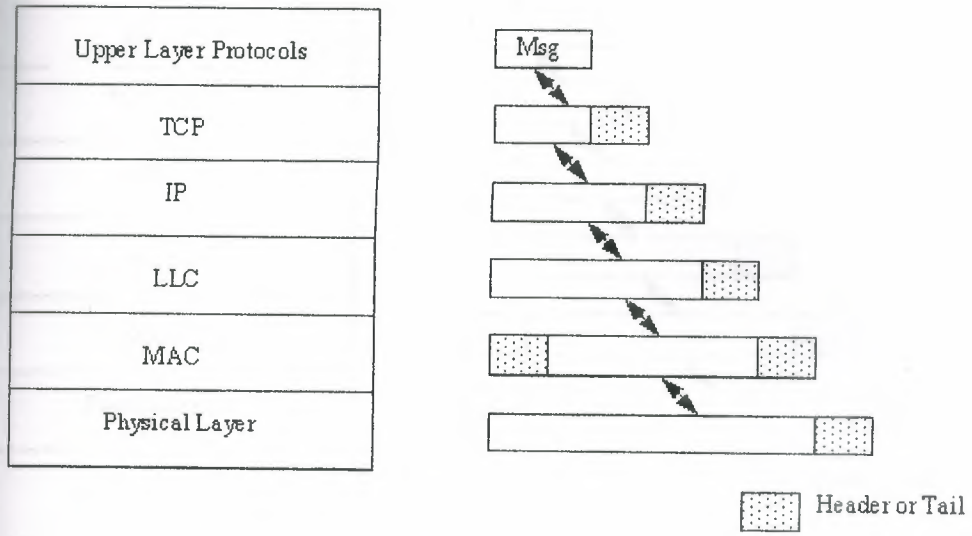

Figure 3.3 TCP/IP with LLC/MAC.

3.1.2 NetBIOS and TCP/IP

A popular PC-oriented network operating system is NetBIOS, which can be cleanly integrated with TCP/IP. Figure 3.4 shows the network architecture for this kind of LAN. NetBIOS resides above the TCP or UDP protocol, although it usually has solid links into that layer (so the two layers cannot be cleanly separated). NetBIOS acts to connect applications together in the upper layers, providing messaging and resource allocation.

Three Internet port numbers are allocated for NetBIOS. These are for the NetBIOS name service (port 137), datagram service (port 138), and session service (port 139). There is also the provision for a mapping between Internet's Domain Name Service (DNS) and the NetBIOS Name Server (NBNS). The NetBIOS Name Server is used to identify PCs that operate in a NetBIOS area. In the interface between NetBIOS and TCP, a mapping between the names is used to produce the DNS name. IP can be configured to run above NetBIOS, eliminating TCP or UDP entirely and running NetBIOS as a connectionless service.

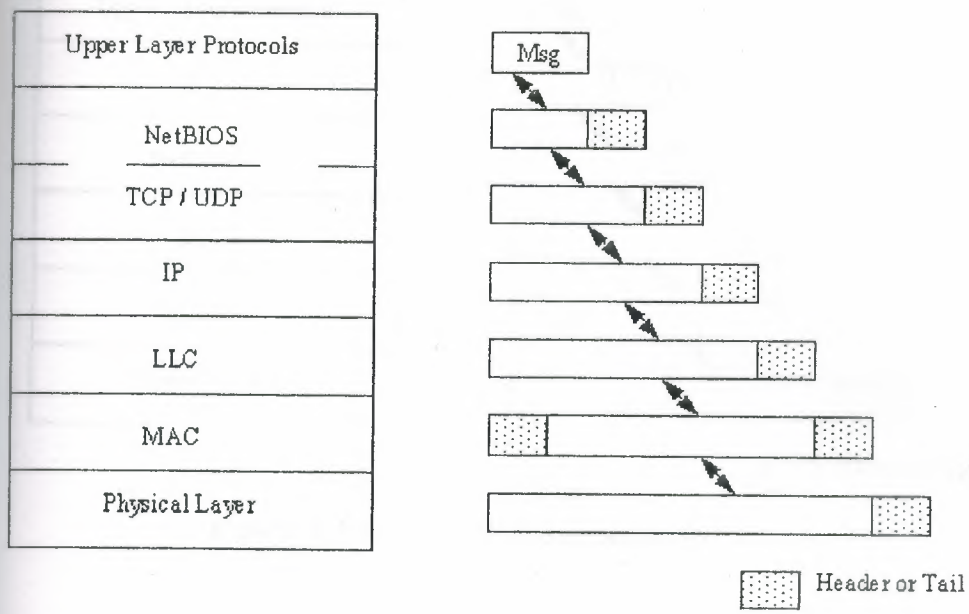

Figure 3.4 The NetBIOS Network architecture.

In this case, NetBIOS takes over the functions of the TCP/UDP layer, and the upper layer protocols must have the data integrity, packet sequencing, and flow control functions. This is shown in Figure 3.5. In this architecture, NetBIOS encapsulates IP datagrams. Strong mapping between IP and NetBIOS is necessary so that NetBIOS packets reflect IP addresses. (To do this, NetBIOS codes the names as IP *nnn.nnn.nnn.nnn*.) This type of network requires that the upper layer protocols (ULPs) handle all the necessary features of the TCP protocol, but the advantage is that the network architecture is simple and efficient. For some networks, this type of approach is well suited, although the development of suitable ULPs can be problematic at times.

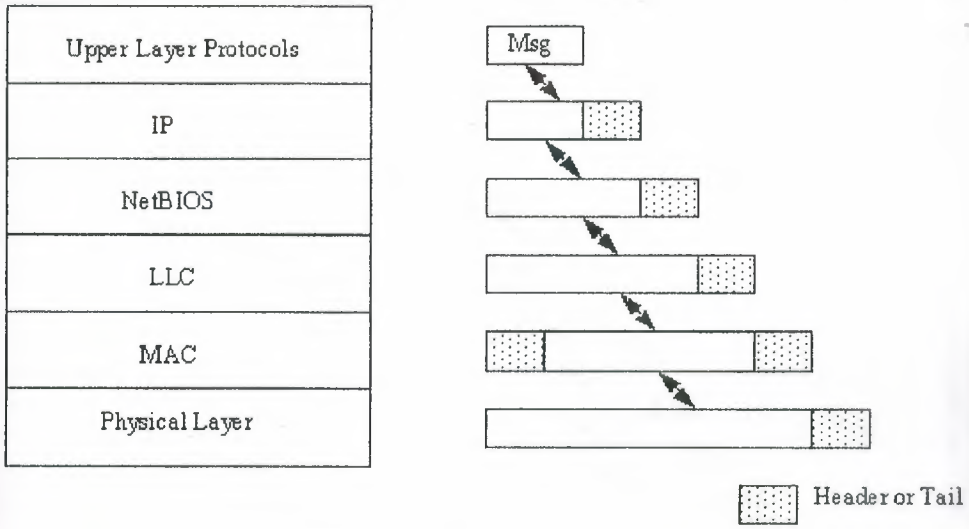

Figure 3.5 Running IP above NetBIOS.

3.1.3 XNS and TCP/IP

XNS appears in several commercial network software packages. XNS can use IP. The Sequenced Packet Protocol (SPP) is above the IP layer, providing some TCP function, although it is not as complete a protocol. In the ULP layer is the Courier protocol, which provides presentation and session layer services.

XNS uses the term *Internet Transport Protocols* to refer to the set of protocols used, including IP. Among the protocols is the Routing Information Protocol (RIP) and an error protocol similar to the Internet Control Message Protocol (ICMP).

3.1.4 IPX and UDP

Novell's NetWare networking product has a protocol similar to IP called the Internet Packet Exchange (IPX), which is based on Xerox's XNS. The IPX architecture is shown in Figure 3.6. IPX usually uses UDP for a connectionless protocol, although TCP can be used when combined with LLC Type 1. The stacking of the layers (with IPX above UDP) ensures that the UDP and IP headers are not affected, with the IPX information encapsulated as part of the usual message process. As with other network protocols, a mapping is necessary between the IP address and the IPX addresses. IPX uses network and host numbers of 4 and 6 bytes, respectively. These are converted as they are passed to UDP. It is possible to reconfigure the network to use IPX networks running TCP instead of UDP and substituting the connectionless LLC Type 1 protocol.

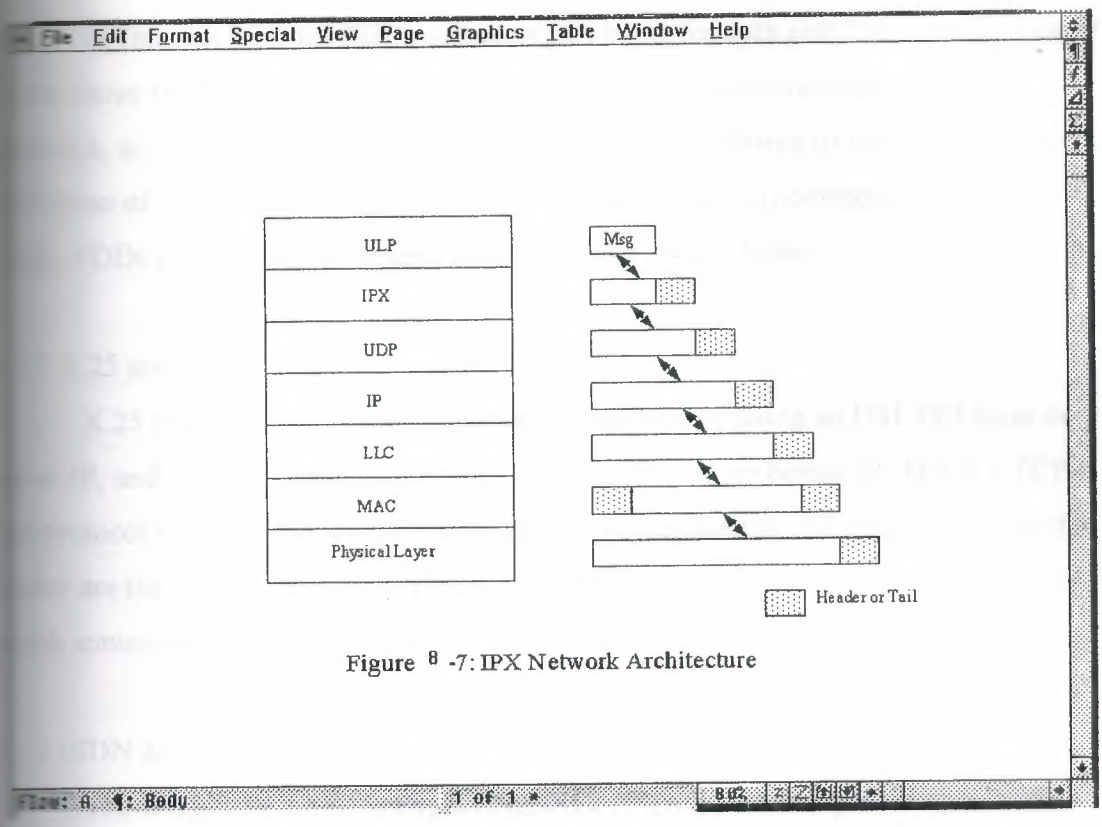

Figure 3.6 The IPX Network architecture.

3.1.5 ARCnet and TCP/IP

ARCnet is widely used for LANs and has an Internet RFC for using it with IP. The architecture is similar to that of the IPX-based network but with ARCnet replacing IPX. Messages passed down from IP are encapsulated into ARCnet datagrams.

3.1.6 FDDI Networks

The Fiber Distributed Data Interface (FDDI) is an ANSI-defined high-speed network that uses fiber-optic cable as a transport medium. FDDI is gaining string support because of the high throughout that can be achieved. For TCP/IP, FDDI uses a layered architecture like the other networks discussed. FDDI differs slightly from other media in that there are two sublayers for the physical layer. FDDI's addressing scheme is similar to other Ethernet networks, requiring a simple mapping, as seen with the Ethernet system. IP and ARP can both be used over FDDI. IP is used with the LLC Type 1 connectionless service.

The frame size for FDDI is set to 4,500 bytes, including the header and other framing information. After that is taken into account, there are 4,470 bytes available for

(The Internet RFC for FDDI defines 4,096 bytes for data and 256 bytes for header above the MAC layer.) This large packet size can cause problems for some gateways, so routing for FDDI packets must be carefully chosen to prevent truncation or corruption of the packet by a gateway that can't handle the large frame size. In case of doubt, FDDI packets should be reduced in size to 576 data bytes.

3.1.7 X.25 and IP

X.25 networks modify the network architecture by using an OSI TP4 layer on top of IP, and the X.25 Packet Layer Procedures (PLP) layer below IP. TP4 is a TCP-like protocol that does not use port identifiers. The destination and source fields in the header are the *transport service access points* (TSAPs). TP4 is more complex than TCP, which sometimes works against it.

3.1.8 ISDN and TCP/IP

The Integrated Services Digital Network (ISDN) provides packet-switched TCP/IP networks. The architecture is shown in Figure 3.7. IP is not in the stack because it is usually incorporated into CLNP. (Both TCP and IP can be used with ISDN instead of OSI TP4 and CLNP, but the ISDN versions are optimized for that network.) ISDN uses a more complex architecture than most networks, replacing gateways and routers with *terminal adapters* and *ISDN nodes*. These perform the equivalent functions but have a more rigid (and complex) internal architecture.

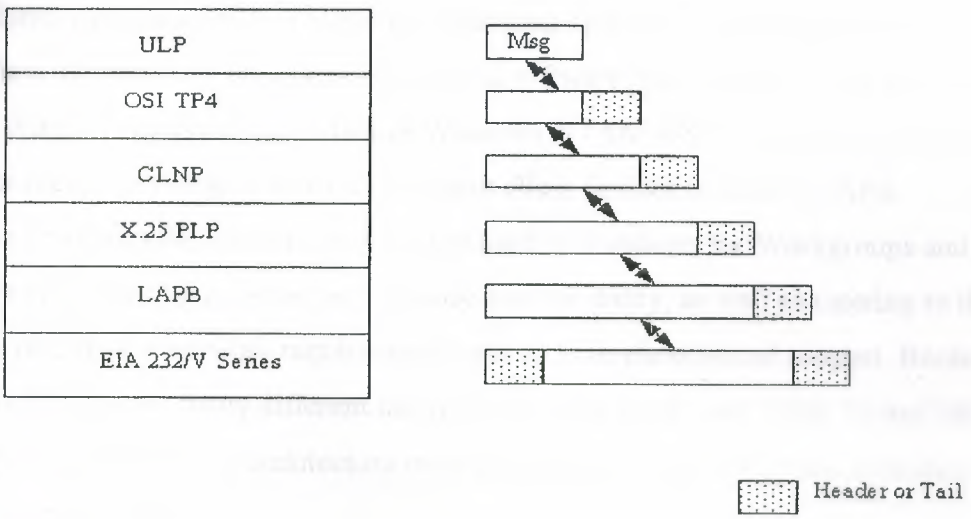

Figure 3.7 The ISDN-based Network architecture.

3.1.9 Switched Multi-Megabit Data Services and IP

The Switched Multi-Megabit Data Services (SMDS) system is a public packet-switched connectionless service that provides high throughput with large packet sizes (up to 9188 data bytes). SMDS uses a subscriber-to-network and network-to-subscriber access mechanism for flow control. SMDS works with IP by interfacing the SMDS to the LLC layer. SMDS using IP supports multiple logical IP subnetworks (LISs), which can be managed separately but treated as a single unit by SMDS. This method requires all the subnetworks to have to same IP address. SMDS uses LLC Type 1 frames.

3.1.10 Asynchronous Transfer Mode (ATM) and BISDN

Two new protocols for high-speed internetworks that are becoming popular are Asynchronous Transfer Mode (ATM) and Broadband ISDN (BISDN). The architecture on the user's machine is similar to the TCP/IP architectures discussed earlier, although additional layers can be added to provide new services, such as video and sound capabilities. The router, gateway, or other device that accesses the high-speed network is more complex as well. Called a *terminal adapter* (as with ISDN), it provides a sophisticated interface between user layers and adaptation layers, which are application-specific. From the terminal adapter, traffic is passed to the ATM service, which provides switching and multiplexing services.

3.1.11 Windows 95 and TCP/IP

Because Windows 95 is supposed to become the dominant operating system on PC machines running a DOS or Windows operating system, it is worth taking a quick look at how Windows 95 integrates networking software into its kernel. The approach used by Windows 95 is similar to that of Windows NT and OS/2, so the knowledge is useful for many operating systems on common client devices in today's LANs. Windows 95 refines the network architecture used in Windows for Workgroups and Windows NT, resulting in better performance and reliability, as well as catering to the demands of different network requirements such as multiple protocol support. Because Windows 95 supports many different network protocols in 16- and 32-bit Virtual Mode Driver (VxD) versions, the architecture must provide the flexibility to accommodate a number of structures.

The Windows 95 architecture is layered; a layered architecture is the most common networking structure (such as OSI and TCP/IP). The network architecture used in Windows 95 is known as Microsoft's Windows Open Services Architecture (WOSA). WOSA was developed to enable applications to work with several different network types, and it includes a set of interfaces designed to enable coexistence of several network components.

The networking software components of Windows 95 are shown in their respective layers in Figure 3.8. Many of the network components are familiar from earlier versions of Windows for Workgroups, Windows NT, or other operating systems and communications protocols. Older 16-bit applications are treated slightly differently, but the principles are the same.

API: The standard Win32 Application Programming Interface (API, the same system used with Windows NT). The API handles remote file operations and remote resources (printers and other devices). The Win32 APIs are used for programming applications.

Multiple Provider Router (MPR): The MPR routes all network operations for Windows 95, as well as implementing network functions common to all network types. Win32 APIs communicate directory with the MPR, although some can be routed straight through. The MPR is a 32-bit protected mode DLL.

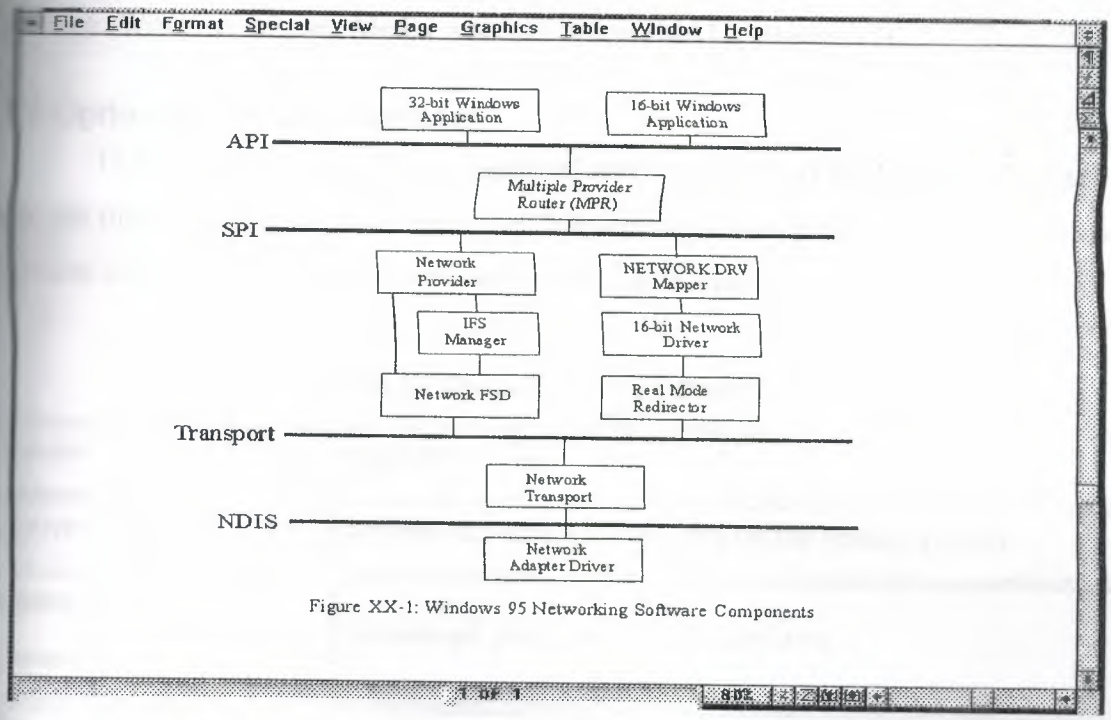

Figure 3.8 The Windows 95 networking software architecture showing the components.

Network Provider: The network provider implements the network service provider interface. Only the MPR can communicate with the network provider. The network provider is a 32-bit protected mode DLL.

IFS Manager: The IFS Manager routes filesystem requests to the proper filesystem driver (FSD). The IFS Manager can be called directly by network providers.

Network Filesystem Driver (FSD): The FSD implements the particular remote filesystem characteristics. The FSD can be used by the IFS Manager when the filesystem of the local and remote machines match. The FSD is a 32-bit protected mode VxD (virtual device driver).

Network Transport: The network transport is a VxD that implements the device-specific network transport protocol. Multiple network transports can be active at a time. The network FSD interfaces with the network transport, usually with a one-to-one mapping, although that is not necessarily the case.

Network Driver Interface Specification (NDIS): A vendor-independent software specification that defines interactions between the network transport and device driver. Windows 95 supports both 32-bit and 16-bit NDIS versions.

Network Adapter Driver: The network adapter driver VxD controls the actual network hardware device. NDIS communicates with the driver, which sends packets over the network. Windows 95 uses Media Access Control (MAC) drivers.

3.2 Optional TCP/UDP Services

TCP/UDP offers a number of optional services that users and applications can use. All these optional services have strict definitions for their protocols. These optional services and their assigned port numbers are shown in Table 3.1.

Table 3.1 Optional TCP/UDP services.

Service	*Port*	*Description*
Active Users	11	Returns the names of all users on the remote system
Character Generator	19	Returns all printable ASCII characters
Daytime	13	Returns the date and time, day of the week, and month of the

		year
Discard	9	Discards all received messages
Echo	7	Returns any messages
Quote of the Day	17	Returns a quotation
Time	37	Returns the time since January 1, 1900 (in seconds)

3.2.1 Active Users

The Active Users service returns a message to the originating user that contains a list of all users currently active on the remote machine. The behavior of the TCP and UDP versions is the same. When requested, the Active Users service monitors port 11 and, upon establishment of a connection, responds with a list of the currently active users and then closes the port. UDP sends a datagram, and TCP uses the connection itself.

3.2.2 Character Generator

The Character Generator service is designed to send a set of ASCII characters. Upon receipt of a datagram (the contents of which are ignored), the Character Generator service returns a list of all printable ASCII characters. The behavior of the TCP and UDP versions of the Character Generator are slightly different. The TCP Character Generator monitors port 19, and upon connection ignores all input and sends a stream of characters back until the connection is broken. The order of characters is fixed. The UDP Character Generator service monitors port 19 for an incoming datagram (UDP doesn't create connections) and responds with a datagram containing a random number of characters. Up to 512 characters can be sent. Although this service might seem useless, it does have diagnostic purposes. It can ensure that a network can transfer all 95 printable ASCII characters properly, and it can also be used to test printers for their capability to print all the characters.

3.2.3 Daytime

The Daytime service returns a message with the current date and time. The format it uses is the day of the week, month of the year, day of the month, time, and the year. Time is specified in a *HH:MM:SS* format. Each field is separated by spaces to enable parsing of the contents. Both TCP and UDP versions monitor port 13 and, upon receipt of a datagram, return the message. The Daytime service can be used for several purposes, including setting system calendars and clocks to minimize variations. It also can be used by applications.

3.2.4 Discard

The Discard service simply discards everything it receives. TCP waits for a connection on port 9, whereas UDP receives datagrams through that port. Anything incoming is ignored. No responses are sent. The Discard service might seem pointless, but it can be useful for routing test messages during system setup and configuration. It can also be used by applications in place of a discard service of the operating system (such as /dev/null in UNIX).

3.2.5 Echo

The Echo service returns whatever it receives. It is called through port 7. With TCP, it simply returns whatever data comes down the connection, whereas UDP returns an identical datagram (except for the source and destination addresses). The echoes continue until the port connection is broken or no datagrams are received. The Echo service provides very good diagnostics about the proper functioning of the network and the protocols themselves. The reliability of transmissions can be tested this way, too. Turnaround time from sending to receiving the echo provides useful measurements of response times and latency within the network.

3.2.6 Quote of the Day

The Quote of the Day service does as its name implies. It returns a quotation from a file of quotes, randomly selecting one a day when a request arrives on port 17. If a source file of quotations is not available, the service fails.

12.7 Time

The Time service returns the number of seconds that have elapsed since January 1, 1990. Port 37 is used to listed for a request (TCP) or receive an incoming datagram (UDP). When a request is received, the time is sent as a 32-bit binary number. It is up to the receiving application to convert the number to a useful figure. The Time service is often used for synchronizing network machines or for setting clocks within an application.

12.8 Using the Optional Services

The optional services can be accessed from an application. Users can directly access their service of choice (assuming it is supported) by using Telnet. A simple example is shown here:

```
$ telnet merlin 7

Trying...

Connected to merlin.tpci.com

Escape character is '^T'.

This is a message

This is a message

Isn't this exciting?

Isn't this exciting?

<Ctrl+T>

$ telnet merlin 13

Trying...

Connected to merlin.tpci.com

Escape character is '^T'.
```

```
Jun 21 10:16:45 1994

Connection closed.

$ telnet merlin 19

"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[[\]^_abcdefg

"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[[\]^_abcdefgh

#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[[\]^_abcdefghi

$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[[\]^_abcdefghij

%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[[\]^_abcdefghijk

&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[[\]^_abcdefghijkl

'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[[\]^_abcdefghijklm

<Ctrl+T>

$
```

In this example, a connection to port 7 starts an Echo session. Everything typed by the user is echoed back immediately, unchanged. Then a connection to port 13 provides the Daytime service, showing the current date and time. The connection is broken by the service once the data is sent. Finally, the Character Generator is started. Both the Echo and Character Generator services were terminated with the Telnet break sequence of Ctrl+T.

3.3 Setting Up a Sample TCP/IP Network: Servers

Here, we look at how to set up four different types of servers: a Santa Cruz Operation (SCO) OpenServer 5 machine, a Linux machine, a Windows NT machine, and a Sun SPARCstation 5. All four servers are connected to the sample network, and any of them can be accessed by a client machine or other servers.

3.3.1 The Sample Network

We designed a dedicated TCP/IP network to show the steps we must follow to set up, configure, and test a TCP/IP implementation. The sample network relies on several servers, although many networks have only one. The sample network has four servers and three clients. Each of the seven machines on the network has its own name and IP address. For this sample network, the IP address mask has been randomly chosen as 147.120. The sample network configuration is shown in Figure 3.9.

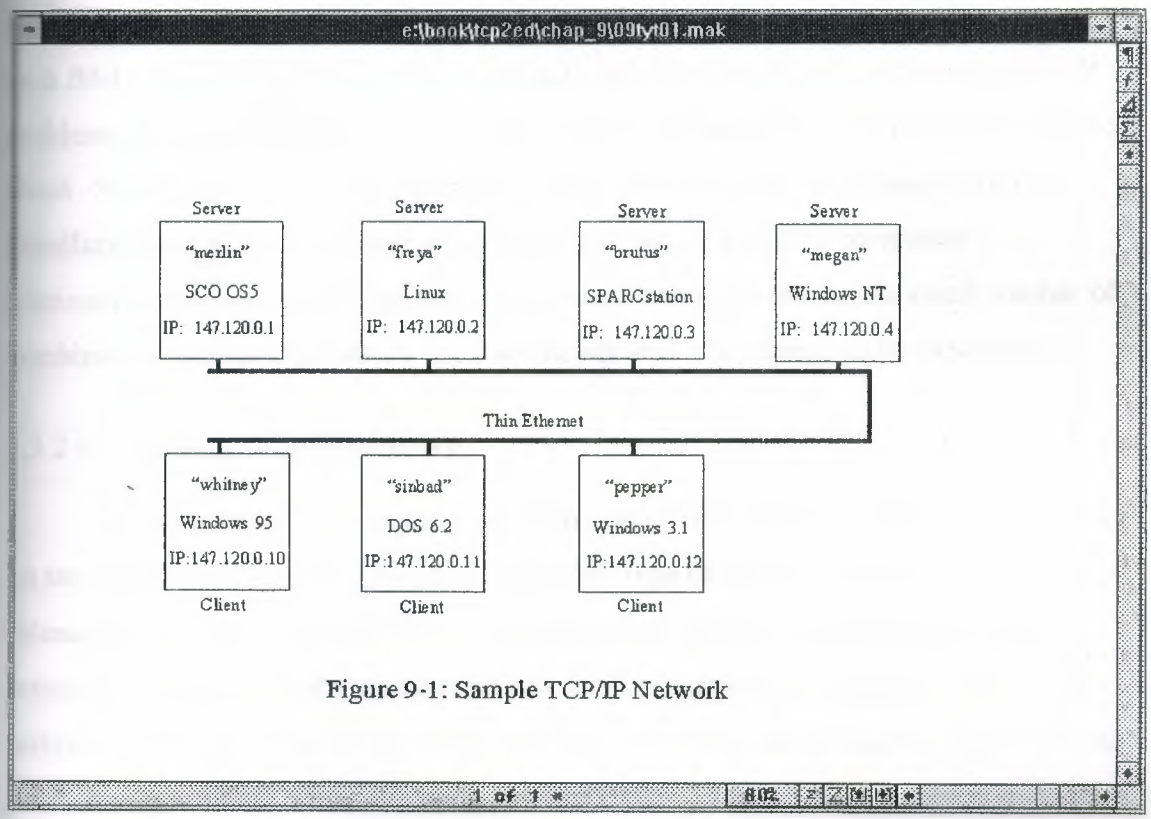

Figure 9-1: Sample TCP/IP Network

Figure 3.9 The sample TCP/IP network.

The physical setup of the network is undertaken first. It involves installing a network interface card in each machine (except the SPARCstation, which has the

network card as part of the motherboard). On each system we must ensure that any jumpers for interrupt vectors and memory I/O addresses do not conflict with any other card on that system. (Some of the cards are software programmable; some are set by jumpers or DIP switches.) All the boards used in this system are from different manufacturers to show the independent nature of the TCP/IP network. Cable must be run between all the machines, connecting the network interface cards together. In the case of Ethernet, the cables must be properly terminated. The sample network uses thin Ethernet, which closely resembles television coaxial cable. BNC Thin Ethernet connectors resemble a T, with cables attached to both ends of the T and the stem connected to the network card. Two of the machines form the ends of the cable and require a terminating resistor as part of their T. The SPARCstation normally uses an RJ45 connector.

To test the physical network, it is easiest to wait until a couple of machines have had their basic software configuration completed. All the machines on the network do not have to be active, as long as the network cable is contiguous from end to end and each BNC connector is attached to a network card to provide electrical termination. If problems are found when the network is tested, the physical network is the first item to check. Some network monitoring devices can supply integrity information prior to installing the network, but these devices are not usually available to system administrators who are just beginning their installation, or who have a small number of machines to maintain (primarily because the network testers tend to be expensive).

3.3.2 Configuring TCP/IP Software

The discussion here applies equally to the UNIX, Windows, and DOS machines on the sample network (as it would to any other type of machine, such as a Macintosh). Filenames can change with different operating systems, but the general approach *remains valid. Most operating systems and TCP/IP software packages provide several* utilities, including menu-driven scripts that help automate the installation process of the TCP/IP applications. Some operating systems (notably older UNIX systems) still require manual configuration of several files using a text editor. To configure TCP/IP software properly, we must know several pieces of information before we start. The necessary information we need for each machine on the network follows:

- **Domain name:** The name the entire network will use.
- **System name:** The unique name of each local machine.
- **IP address:** The full address of each machine.
- **Driver type:** Each interface to the network must be associated with a device driver, instructing the operating system how to talk to the device.
- **Broadcast address:** The address used for network-wide broadcasts.
- **Netmask:** The network mask that uniquely identifies the local network.
- **Hardware network card configuration information:** The interrupt vector and memory address of the network card.

The system domain name is necessary if the network is to be connected to other machines outside the local network. Domain names can be invented by the system administrator. If, however, the network is to interface with Internet or one of its service providers, the domain name should be approved by the Internet Network Information Center (InterNIC). Creating and registering a new domain is as simple as filling out a form (and recently, paying a small administration fee). Domain names usually reflect the company name, with the extension identifying the type of organization. The sample network uses the name tpci.com.

The machine name is used for symbolic naming of a machine instead of forcing the full IP address to be specified. The system name must be unique on the local network. Other networks might have machines with the same name, but their network masks are different, so there is no possible confusion during packet routing. In most cases, system names are composed of eight characters (or less) and are usually all lowercase characters (in keeping with UNIX tradition for lowercase). The system name can be a mix of characters and numbers. Larger organizations tend to number their machines, and small companies give their machines more familiar names. The device driver instructs the operating system how to communicate with the network interface (usually either a network card or a serial port). Each interface has its own specific device driver. Most operating systems have device drivers included in their distribution software, although some require software supplied with the network card. Generic drivers are available for most network cards on bulletin board systems.

Most network cards come with default settings that might conflict with other cards in the system. Users must carefully check for conflicts, resorting to a diagnostic program if available. UNIX users have several utilities available, depending on the

operating system. SCO UNIX and most System V Release 4 operating systems have the utility hwconfig, which shows the current hardware configuration. The following example shows the hwconfig output and the output from the command with the -h option to provide long formatting with headers (making it is easier to read):

```
$ hwconfig

name=fpu vec=13 dma=- type=80387

name=serial base=0x3F8 offset=0x7 vec=4 dma=- unit=0 type=Standard
nports=1

name=serial base=0x2F8 offset=0x7 vec=3 dma=- unit=1 type=Standard
nports=1

name=floppy base=0x3F2 offset=0x5 vec=6 dma=2 unit=0 type=96ds15

name=floppy vec=- dma=- unit=1 type=135ds18

name=console vec=- dma=- unit=vga type=0 12 screens=68k

name=adapter base=0x2C00 offset=0xFF vec=11 dma=- type=arad ha=0 id=7
fts=st

name=nat base=0x300 offset=0x20 vec=7 dma=- type=NE2000
addr=00:00:6e:24:1e:3e

name=tape vec=- dma=- type=S ha=0 id=4 lun=0 ht=arad

name=disk vec=- dma=- type=S ha=0 id=0 lun=0 ht=arad fts=stdb

name=Sdsk vec=- dma=- cyls=1002 hds=64 secs=32

$

$ hwconfig -h

device          address     vec  dma  comment
```

```
======          =======     ===   ===   =======

fpu                -         13    -    type=80387

serial         0x3f8-0x3ff    4    -    unit=0 type=Standard nports=1

serial         0x2f8-0x2ff    3    -    unit=1 type=Standard nports=1

floppy         0x3f2-0x3f7    6    2    unit=0 type=96ds15

floppy             -          -    -    unit=1 type=135ds18

console            -          -    -    unit=vga type=0 12 screens=68k

adapter       0x2c00-0x2cff  11    -    type=arad ha=0 id=7 fts=st

nat            0x300-0x320    7    -    type=NE2000
addr=00:00:6e:24:1e:3e

tape               -          -    -    type=S ha=0 id=4 lun=0 ht=arad

disk               -          -    -    type=S ha=0 id=0 lun=0 ht=arad
fts=stdb

Sdsk               -          -    -    cyls=1002 hds=64 secs=32
```

This output is from the SCO UNIX servers set up for the sample network. It has the network Ethernet card already configured as device nat, which uses IRQ 7 (shown under the vec or interrupt vector column). The nat line also shows the memory address as 300–320 (hexadecimal) and the device driver as NE2000 (a Novell NetWare-compatible driver). The address and vec columns show no conflicts between the settings used for the Ethernet card and other devices on the system. (The adapter entry is for a high-speed SCSI-2 card, which controls both the tape and the Sdsk device, the primary SCSI hard drive. All other entries should be self-explanatory.) DOS users can use the Microsoft Diagnostic utility, MSD.EXE, or one of several third-party tools such as Central Point PC Tools or The Norton Utilities to display IRQ vectors and memory addresses in use by the system. Some software even indicates which vectors and addresses are available for use.

There is no need to have the same IRQ and memory address for each card on the network, because the network itself doesn't care about these settings. The IRQ and memory addresses are required for the machine to communicate with the network interface card only. The sample network used a different IRQ and memory address for each machine.

IRQ and memory addresses are usually set on the network interface card itself using either jumpers on pins or a DIP-switch block. The documentation accompanying the card should provide all the information necessary for setting these values. Some recently introduced network interface cards can be configured through software, enabling the settings to be changed without removing the card from the system. This can be very handy when a user is unsure of the best settings for the card.

The IP address is a 32-bit number that must be unique for each machine. If the network is to be connected to the Internet, the IP address must be assigned by the NIC. Even if no access to the Internet is expected, arbitrarily assigning an IP address can cause problems when messages are passed with other networks. If the network is not connected to the outside world, a system administrator can ignore the NIC's numbering system and adopt any IP address. It is worthwhile, however, to consider future expansion and connection to other networks. The NIC has four classes of IP addresses in use depending on the size of the network. Each class has some addresses that are restricted. These are shown in Table 3.2. Most networks are Class B, although a few large corporations require Class A networks.

Table 3.2 The NIC IP address classes.

Class	*Network Mask Bytes*	*Number of Hosts per Network*	*Valid Addresses*
A	1	16,777,216	1.0.0.1 to 126.255.255.254
B	2	65,534	128.0.0.1 to 191.255.255.254
C	3	254	224.0.0.0 to 255.255.255.254
D	reserved		

The network mask is the IP address stripped of its network identifiers, leaving only the local machine address. For a Class A network, this strips one byte, whereas a Class B network strips two bytes (leaving two). The small Class C network strips three bytes as the network mask, leaving one byte to identify the local machine (hence the limit of 254 machines on the network). The sample network is configured as a Class B machine with the randomly chosen IP address network mask of 147.120 (not NIC-assigned).

The broadcast address identifies packets that are to be sent to all machines on the local network. Because a network card usually ignores any incoming packets that don't have its specific IP address in them, a special broadcast address can be set that the card can intercept in addition to locally destined messages. The broadcast address has the host portion (the local machine identifiers) set to either all 0s or all 1s, depending on the convention followed. For convenience, the broadcast address's network mask is usually the same as the local network mask.

Broadcast addresses might seem simple because there are only two possible settings. Such addresses, however, commonly cause problems because conflicting settings are used on a network. BSD UNIX used the convention of all 0s for releases 4.1 and 4.2, whereas 4.3BSD and SVR4 (System V Release 4) UNIX moved to all 1s for the broadcast address. The Internet standard specifies all 1s as the broadcast address. If problems are encountered on the network with broadcasts, check all the configurations to ensure they are using the same setting. The sample network uses an all 1s mask for its broadcast address.

The steps followed for configuring TCP/IP are straightforward, generally following the information required for each machine. The configuration steps are as follows:

- **Link drivers:** TCP/IP must be linked to the operating system's kernel or loaded during the boot stage to enable TCP/IP.
- **Add host information:** Provide a list of all machines (hosts) on the network (used for name resolution).
- **Establish routing tables:** Provide the information for routing packets properly if name resolution isn't sufficient.

- **Set user access:** Configure the system to enable access in and out of the network, as well as establishing permissions.
- **Remote device access:** Configure the system for access to remote printers, scanners, CD-ROM carousels, and other shared network devices.
- **Configure the name domain server:** If using a distributed address lookup system such as Berkeley Internet Name Domain Server (BIND) or NIS, complete the name server files. (This step is necessary only if you are using BIND or a similar service.)
- **Tune system for performance:** Because a system running TCP/IP has different behavior than one without TCP/IP, some system tuning is usually required.
- **Configure NFS:** If the Network File System (NFS) is to be used, configure both the file system and the user access.
- **Anonymous FTP:** If the system is to enable anonymous FTP access, configure the system and public directories for this service.

We will use these steps (not necessarily in the sequence given) as the individual machines on the network are configured. The processes are different with each operating system, but the overall approach remains the same.

3.4 Setting Up a Sample TCP/IP Network: DOS and Windows Clients

Here we configure some clients for the network. The clients communicate with the server through a TCP/IP stack loaded on each machine. We configure three clients: one DOS, one Windows 3.*x*, and one Windows 95.

3.4.1 DOS-Based TCP/IP: ftp Software's PC/TCP

PC/TCP runs under both DOS and Windows. It lets a user perform all the TCP/IP functions, such as ftp and telnet, and includes software for several members of the TCP family of protocols, including SNMP. Other machines can also access a PC running PC/TCP, copying its files (assuming access has been granted). PC/TCP can run TCP/IP as the sole network protocol on the PC, or it can piggy-back on top of other networks, such as Windows for Workgroups (NetBEUI and NetBIOS) or Novell NetWare (IPX/SPX). The sample network we are configuring is TCP/IP-based, so PC/TCP is installed to run on that network protocol only. However, because it would be useful to be able to run Windows for Workgroups over the network between the DOS

and Windows 3.11 machines, the installation process we take is designed so that both NetBEUI and TCP/IP can reside simultaneously on the network.

The sample network we are installing is configured to enable both PC/TCP and Windows for Workgroups to coexist using NDIS drivers. This results in two software stacks—one for PC/TCP and one for Windows for Workgroups—coexisting and communicating with the NDIS driver. This structure is shown in Figure 3.10.

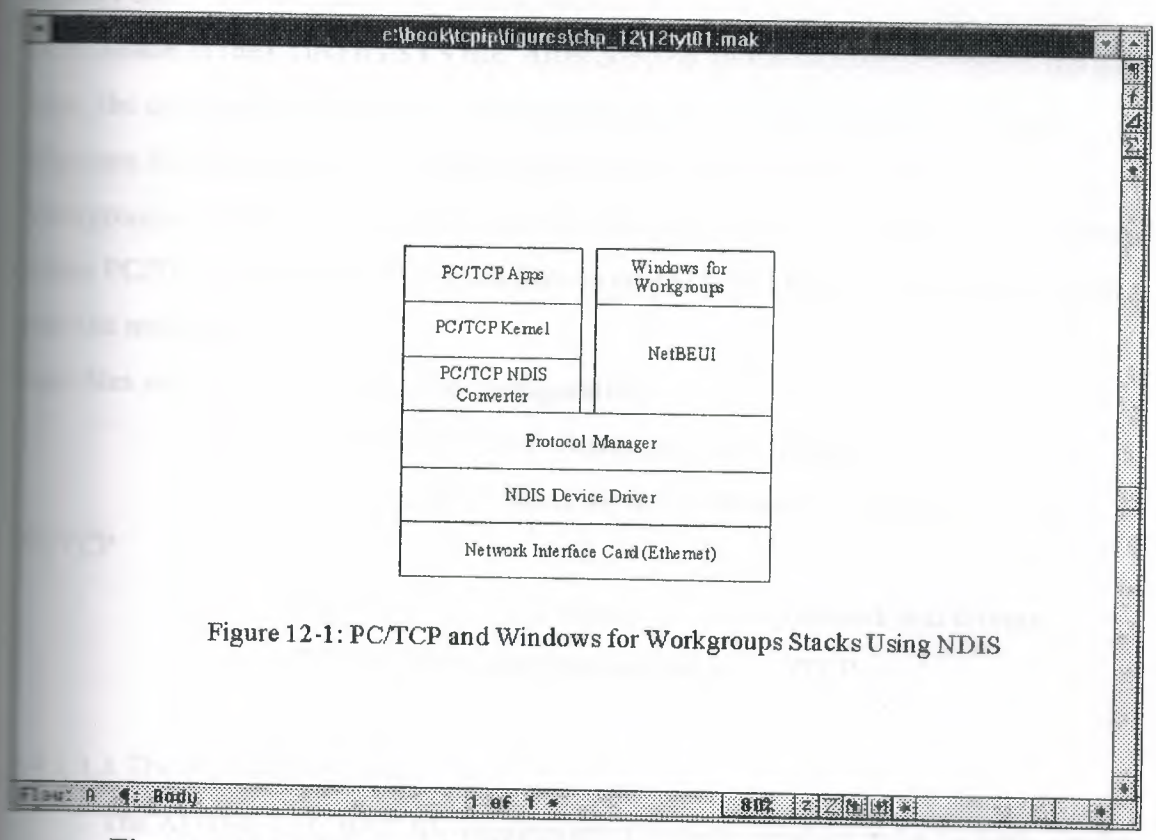

Figure 12-1: PC/TCP and Windows for Workgroups Stacks Using NDIS

Figure 3.10 PC/TCP and Windows for Workgroups stacks using NDIS.

PC/TCP uses a kernel that is loaded into memory when DOS boots. The kernel is a Terminate and Stay Resident (TSR) program. To ensure that the network is available at all times, the kernel load command is usually added to the AUTOEXEC.BAT file. The sample network uses a kernel called ETHDRV.EXE, which is the Ethernet driver supplied with PC/TCP. (A different kernel must be used if the network is IEEE 802.3 Ethernet, which differs from the normal DIX Ethernet.) In addition, an NDIS Converter must be loaded in the AUTOEXEC.BAT file as a device driver to provide NDIS-format packets to the protocol manager.

3.4.1.1 Installing PC/TCP

PC/TCP includes an automated installation procedure that copies the distribution media to the hard disk and sets up some of the configuration files. Installation of PC/TCP requires the same basic information as TCP/IP under UNIX: the device driver, the system's name and IP address, and the names and IP addresses of other systems to be accessed. The process begins with a properly installed network card. The IRQ and memory address of the card must be known, and a device driver for it must be present for inclusion in the CONFIG.SYS file. After copying all the distribution files to the hard drive, the configuration can begin. The sample machine is running DOS 6.22 and Windows for Workgroups 3.11. When installing PC/TCP with Windows for Workgroups, the Windows network must be installed, configured, and running properly before PC/TCP modifies the Windows files to enable both DOS and Windows to work over the network.

Four files are involved in the initial configuration:

- AUTOEXEC.BAT: Starts the PC/TCP kernel
- CONFIG.SYS: Starts the device drivers for the network and PC/TCP
- PROTOCOL.INI: Defines the type of network and drivers
- PCTCP.INI: Kernel parameters for PC/TCP

3.4.1.1.1 The AUTOEXEC.BAT File

The AUTOEXEC.BAT file requires environment variables to be properly set for PC/TCP and two instructions added to the file. One instruction starts the network and the other loads the Ethernet driver. The sample machine already had Windows for Workgroups installed, so a line in the AUTOEXEC.BAT file reads

```
C:\WINDOWS\NET START
```

This line starts the network. The NET START command can remain in place or be replaced with a PC/TCP command called NETBIND, which accomplishes the same thing for NDIS drivers. If both commands are in the AUTOEXEC.BAT file, an error message results when the second network startup command is executed.

After the NET START or NETBIND command, the following line must be added to the AUTOEXEC.BAT file:

```
C:\PCTCP\ETHDRV
```

This starts the PC/TCP Ethernet driver. If another network system is being used, this would be replaced with the device driver for that network (such as IEEEDRV for IEEE 802.3 Ethernet or SLPDRV for SLIP). It is useful to define two environment variables in the AUTOEXEC.BAT file for the PC/TCP software to use when searching for file. One is a simple addition to the PATH command, adding the PCTCP installation directory to the search path. The second is an environment variable that points to the PCTCP.INI file. The two declarations look like this:

```
SET PCTCP=C:\PCTCP\PCTCP.INI

SET PATH=C:\PCTCP;%PATH%
```

Therefore, on the DOS machine, the completed AUTOEXEC.BAT file should have one of the following four-line combinations in it:

```
SET PCTCP=C:\PCTCP\PCTCP.INI

SET PATH=C:\PCTCP;%PATH%

C:\WINDOWS\NET START

C:\PCTCP\ETHDRV
```

or

```
SET PCTCP=C:\PCTCP\PCTCP.INI

SET PATH=C:\PCTCP;%PATH%

C:\PCTCP\NETBIND

C:\PCTCP\ETHDRV
```

When these lines are executed during the system boot process, the system displays status messages when each command is completed. The NETBIND command displays this message if it loads successfully:

```
MS-DOS LAN Manager v2.1 Netbind

Microsoft Netbind version 2.1
```

A third line might display a status message about the interrupt vector used by the system. If NETBIND couldn't load correctly, it generates a message like this:

```
MS-DOS LAN Manager v2.1 Netbind

Error: Making PROTMAN IOCTL call.
```

This usually is generated when the network is already running (such as from issuing a NET START command before the NETBIND command; we might recall that only one of these two should be in the AUTOEXEC.BAT file).

The ETHDRV command displays a message with status information when it loads successfully. It looks like this:

```
MAC/DIS converterFTP Software PC/TCP Resident Module 2.31   01/07/94
12:38

Copyright  1986-1993 by FTP Software, Inc.  All rights reserved.

Patch level 17637

Patch time: Fri Jan 07 14:25:09 1994

Kernel interrupt vector is 0x61

Code Segment occupies 49.0K of conventional memory

Data Segment occupies 19.5K of conventional memory

Packet Driver found at vector 0x60

        name:
```

```
        version: 30, class: 1, type: 57, functionality: 6

ifcust (PC/TCP Class 1 packet driver - DIX Ethernet) initialized

5 free packets of length 1514, 5 free packets of length 160

The Resident Module occupies 68.7K of conventional memory
```

If there is an error when the ETHDRV program loads, it generates an error message (of varying utility for debugging purposes). A sample error is shown here:

```
FTP Software PC/TCP Resident Module 2.31     01/07/94 12:38

Copyright  1986-1993 by FTP Software, Inc.  All rights reserved.

Patch level 17637

Patch time: Fri Jan 07 14:25:09 1994

PC/TCP is already loaded (interrupt 0x61). Use 'inet unload' to unload
it.
```

This error occurred because a PC/TCP driver had been loaded prior to the ETHDRV command.

3.4.1.1.2 The CONFIG.SYS File

The CONFIG.SYS file has to have drivers loaded for the protocol manager, the NDIS packet converter, and the network card driver. Systems running Windows for Workgroups might require additional drivers. The CONFIG.SYS file must have an entry setting the number of files open at one time to at least 20. If this doesn't exist, PC/TCP crashes. Add this line:

```
FILES=20
```

to the CONFIG.SYS file. Depending on the amount of memory available, the number could be readily increased. With 8MB RAM or more, a value of 40 is satisfactory. Numbers above this setting tend to be counter-productive because RAM is wasted for no reason.

The protocol manager is supplied as part of Windows for Workgroups, and one is included with the PC/TCP software package. If Windows for Workgroups 3.1 (not 3.11) was already loaded and functional, CONFIG.SYS has a line similar to this:

```
DEVICE=C:\WINDOWS\PROTMAN.DOS /I:C:\WINDOWS
```

The protocol manager is not always used with the Windows for Workgroups 3.11 release because it is included with other drivers within the CONFIG.SYS file (such as IFSHLP.SYS). If there is no protocol manager started at boot time, one should be added from the PC/TCP software. The entry within the CONFIG.SYS file is

```
DEVICE=C:\PCTCP\PROTMAN.DOS \I:C:\PCTCP
```

This loads the PC/TCP protocol manager. The \I at the end of the command tells the driver where to look for files (in this case, the PC/TCP installation directory).
A network card driver should appear next in CONFIG.SYS. This differs for each network card, but for the sample network DOS machine's Intel EtherExpress 16 network card, the line is

```
DEVICE=C:\WINDOWS\EXP16.DOS
```

This loads the EXP16 driver for the Intel network card. This was included with the Windows for Workgroups software, but it is also available as a generic driver. Some machines with Windows for Workgroups already installed might have this command already in the CONFIG.SYS file. The final step is to load the PC/TCP NDIS Packet Converter. The current release of PC/TCP uses a packet converter called DIS_PKT.GUP. The line looks like this:

```
DEVICE=C:\PCTCP\DIS_PKT.GUP
```

The properly configured CONFIG.SYS file for the DOS machine should have these lines in it

```
DEVICE=C:\WINDOWS\PROTMAN.DOS /I:\C:\WINDOWS
```

```
DEVICE=C:\WINDOWS\EXP16.DOS

DEVICE=C:\PCTCP\DIS_PKT.GUP
```

if it is using the Windows for Workgroups protocol manager. It should have the following lines if it is using the PC/TCP protocol manager:

```
DEVICE=C:\PCTCP\PROTMAN.DOS /I:\C:\PCTCP

DEVICE=C:\WINDOWS\EXP16.DOS
DEVICE=C:\PCTCP\DIS_PKT.GUP
```

As noted earlier, the network interface driver (EXP16) is different if our machine does not use the Intel EtherExpress 16 board.

3.4.1.1.3 The PROTOCOL.INI File

Windows for Workgroups has a PROTOCOL.INI file as part of its setup. The file tells the system about the network cards and drivers in use. The PC/TCP PROTOCOL.INI file does the same, but it resides in the PCTCP directory. The contents of the PROTOCOL.INI file are different for each network card and driver configuration. There must be a section labeled [PKTDRV] (all in uppercase) that defines the driver name, the binding to the network card, and any configuration information needed. The sample network's PROTOCOL.INI file looks like this:

```
[PKTDRV]

drivername=PKTDRV$

bindings=MS$EE16

intvec=0x60

[MS$EE16]

DriverName=EXP16$

IOADDRESS=0x360

IRQ=11
```

```
IOCHRDY=Late

TRANSCEIVER=Thin Net (BNC/COAX)
```

3.4.1.1.4 The PCTCP.INI File

The PCTCP.INI file holds the kernel configuration information for PCTCP. In most cases, it can be left as supplied with the software. Tweaking the kernel parameters should be performed only after the network is installed and has been operating properly for a while. The PCTCP.INI file is quite lengthy, and care should be taken to avoid accidental changes, which can render the system inoperative. If the supplied installation script is not used to install PC/TCP, a minimum PCTCP.INI file must be created manually. There are two ways to create the PCTCP.INI file and configure it properly. The first is to use an editor and modify the template file. The alternative is to run the kernel configuration utility KAPPCONF.

A minimum PCTCP.INI file needs to have the software serial number and activation key, the IP address, broadcast address, router address, a subnet mask, and information about the system in general. The minimum PCTCP.INI file would look like this:

```
[pctcp general]

domain                  = tpci.com

host-name               = sinbad

time-zone               = EST

time-zone-offset        = 600

user                    = tparker

[pctcp kernel]

serial-number           = 1234-5678-9012

authentication-key      = 1234-5678-9012
```

```
interface               = ifcust 0

low-window              = 0

window                  = 2048

[pctcp ifcust 0]

broadcast-address       = 255.255.255.255

ip-address              = 147.120.0.11

router                  = 147.120.0.1

subnet-mask             = 255.255.0.0

[pctcp addresses]

domain-name-server      = 147.120.0.1

mail-relay              = 147.120.0.1
```

This configuration assumes that the SCO UNIX server (147.120.0.1) is the primary server for the network. The DOS machine's name (sinbad) and IP address (147.120.0.11) are shown in the PCTCP.INI file. As different features of PC/TCP are enabled (such as SNMP and Kerberos), new sections are added to the PCTCP.INI file.

3.4.1.1.5 The Windows SYSTEM.INI File

If Windows for Workgroups is to be used on the DOS machine and we are going to use the PC/TCP drivers instead of a dedicated Windows for Workgroups TCP/IP package, the Windows for Workgroups SYSTEM.INI file requires modification. The Windows for Workgroups SYSTEM.INI file must be set to use the Windows for Workgroups driver instead of the PC/TCP driver. When the PC/TCP automatic installation process detects a copy of Windows, it makes changes to the SYSTEM.INI file for us. Some of these changes must be checked and modified to enable Windows to boot properly with the PC/TCP drivers. One of the most important changes is the

commenting out of the Windows for Workgroups network driver and its replacement with the PC/TCP driver:

```
network.drv=C:\PCTCP\PCTCPNET.DRV
```

For Windows for Workgroups 3.1, confirm that the SYSTEM.INI file has these three sections, with these commands shown:

```
[boot]

network.drv=wfwnet.drv

[boot.description]

network.drv=Microsoft Windows for Workgroups (version 3.1)

[386Enh]

device=c:\pctcp\vpctcp.386

device=c:\pctcp\wfwftp.386
```

Windows for Workgroups 3.11 has a slightly different SYSTEM.INI. It should look like this:

```
[boot]

network.drv=wfwnet.drv

[boot.description]

network.drv=Microsoft Windows Network (version 3.11)

[386Enh]

device=c:\pctcp\vpctcp.386
```

At the bottom of the Windows for Workgroups SYSTEM.INI file, PC/TCP sometimes adds a block of information that looks like this:

```
[vpctcp]

; These option settings may be added to SYSTEM.INI, in a

;   new section "[vpctcp]".

; The next line tells VPCTCP how much copy space memory to request.

;   It is in units of kilobytes (x1024).  This value is only a bid,
;   as Windows may choose to reduce your allocation arbitrarily.

;   This value should be increased if using Windows applications which

;   call the PC/TCP DLL from another DLL; suggested value in such

;   instances is at least 28.

MinimumCopySpace=12

; The next line tells VPCTCP the segment (paragraph) number of the

;   beginning of memory reserved for devices, BIOS, and upper-

;   memory blocks (which could contain TSRs).  All calls below the

;   PSP of Windows or above this parameter are not processed by

;   the VxD but rather are passed-thru to the kernel untouched.

HiTSRFenceSegment=A000h

; eof
```

3.4.1.2 Windows for Workgroups using NetBIOS

Windows for Workgroups can be set to use IP packets. This requires a NetBIOS driver for both Windows for Workgroups and PC/TCP. The architecture of such as system is shown in Figure 3.11. The Windows for Workgroups packets are sent through PC/TCP's NetBIOS and then into the normal PC/TCP stack.

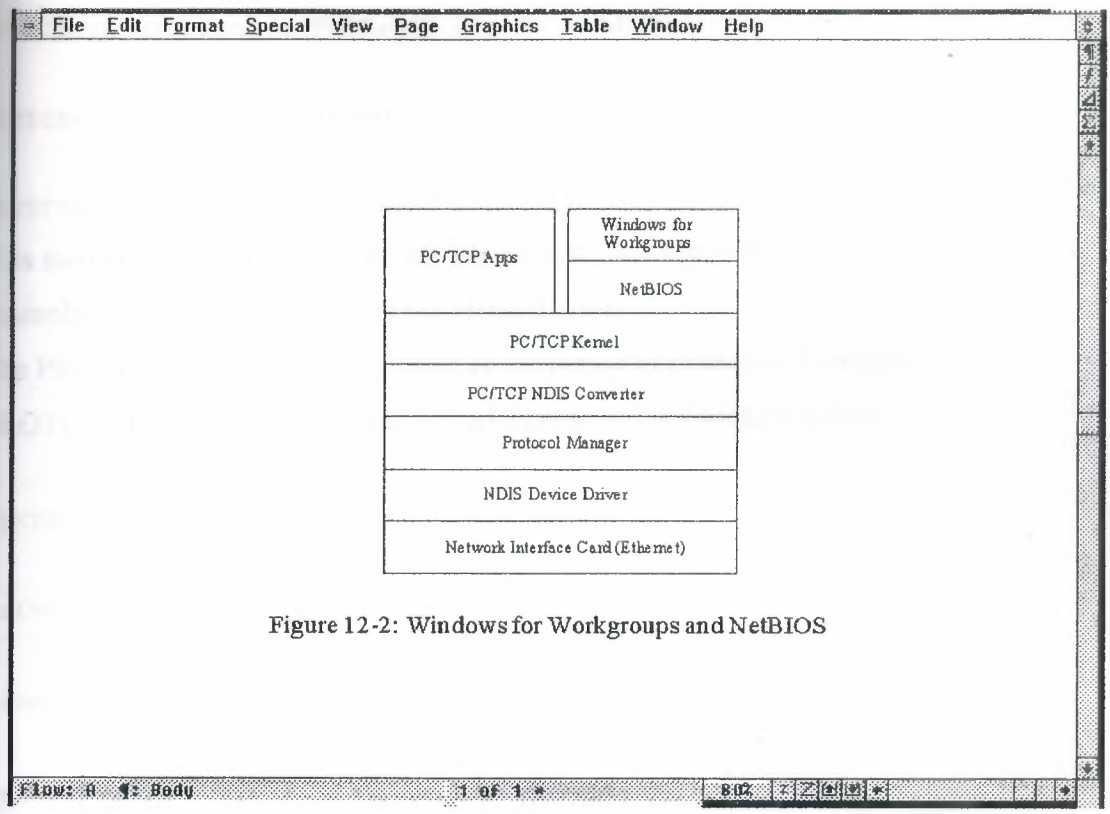

Figure 12-2: Windows for Workgroups and NetBIOS

Figure 3.11 Windows for Workgroups with NetBIOS.

To install Windows for Workgroups in this manner, Windows must first be set up to use the Microsoft LAN Manager option. This is usually a matter of selecting the LAN Manager option from the Network window if it is not already the default setting. The configuration files must also be changed to reflect the new architecture. The AUTOEXEC.BAT file has the network initiation command, the network kernel driver, and a NETBIOS command:

```
C:\WINDOWS\NET START

C:\PCTCP\ETHDRV

C:\PCTCP\NETBIOS.COM
```

A NETBIND can be performed instead of a NET START command, although the latter is preferable. The NETBIOS command must come after the NETBIND or NET START command.

The CONFIG.SYS file is similar to that seen earlier, with the same drivers. A sample CONFIG.SYS file for this type of architecture looks like this:

```
DEVICE=C:\WINDOWS\PROTMAN.DOS /I:\C:\WINDOWS

DEVICE=C:\WINDOWS\EXP16.DOS

DEVICE=C:\PCTCP\DIS_PKT.GUP
```

This starts the protocol manager, the card driver, and the NDIS packet converter. This example uses the Intel EtherExpress 16 card driver.

The PROTOCOL.INI file is the same as the previous example. A sample PROTOCOL.INI file for the Intel EtherExpress 16 card looks like this:

```
[PKTDRV]

drivername=PKTDRV$

bindings=MS$EE16

intvec=0x60

[MS$EE16]

DriverName=EXP16$

IOADDRESS=0x360

IRQ=11

IOCHRDY=Late

TRANSCEIVER=Thin Net (BNC/COAX)
```

Finally, the SYSTEM.INI file requires that the Windows for Workgroups network driver be used and not the PC/TCP network driver. This might require editing the SYSTEM.INI file, as noted earlier. The SYSTEM.INI file should contain the following lines:

```
[boot]

network.drv=wfwnet.drv
```

```
[boot.description]

network.drv=Microsoft Windows for Workgroups (version 3.1)

[386Enh]

device=c:\pctcp\vpctcp.386

device=c:\pctcp\wfwftp.386

TimerCriticialSection=50000
```

The last line in the [386Enh] section might have to be added manually. The version number in the [boot.description] section changes to (version 3.11) with the later version of Windows for Workgroups.

3.4.1.3 Testing PC/TCP

After making all the changes previously mentioned, the DOS machine is rebooted for testing. If no error messages are displayed when the new commands are executed, the system is ready for testing the TCP/IP protocol stack. The simplest test is to use ping to ensure that the TCP/IP software is talking to the local machine, then use it to test the remote machines. Machine name information for other machines hasn't yet been added to the PC/TCP DOS system, so IP addresses must be used with ping. The following is an example of a ping command for the local machine (147.120.0.11), the SCO UNIX server (147.120.0.1), and the Windows 95 machine (147.120.0.10) on the sample network (which has not yet been installed and hence should not communicate):

```
C:\> ping 147.120.0.11

host responding, time = 25 ms

Debugging information for interface ifcust  Addr(6): 00 aa 00 20 18 bf

interrupts: 0 (2 receive, 0 transmit)

packets received: 2, transmitted: 3

receive errors: 0, unknown types: 0
```

```
     runts: 0, aligns: 0, CRC: 0, parity: 0, overflow: 0
     too big: 0, out of buffers: 0, rcv timeout: 0, rcv reset: 0
transmit errors: 0
     collisions: 0, underflows: 0, timeouts: 0, resets: 0
     lost crs: 0, heartbeat failed: 0
ARP statistics:
arps received: 1 (0 requests, 1 replies)
     bad: opcodes: 0, hardware type: 0, protocol type: 0
arps transmitted: 2 (2 requests, 0 replies)
5 large buffers; 4 free now; minimum of 3 free
5 small buffers; 5 free now; minimum of 4 free
C:\>
C:\> ping 147.120.0.1
host responding, time = 25 ms
Debugging information for interface ifcust  Addr(6): 00 aa 00 20 18 bf
interrupts: 0 (5 receive, 0 transmit)
packets received: 5, transmitted: 6
receive errors: 0, unknown types: 0
     runts: 0, aligns: 0, CRC: 0, parity: 0, overflow: 0
     too big: 0, out of buffers: 0, rcv timeout: 0, rcv reset: 0
```

```
transmit errors: 0
     collisions: 0, underflows: 0, timeouts: 0, resets: 0
     lost crs: 0, heartbeat failed: 0
ARP statistics:
arps received: 2 (0 requests, 2 replies)
     bad: opcodes: 0, hardware type: 0, protocol type: 0
arps transmitted: 3 (3 requests, 0 replies)
5 large buffers; 4 free now; minimum of 3 free
5 small buffers; 5 free now; minimum of 4 free
C:\>
C:\> ping 147.120.0.10
ping failed: Host unreachable: ARP failed
Debugging information for interface ifcust  Addr(6): 00 aa 00 20 18 bf
interrupts: 0 (5 receive, 0 transmit)
packets received: 5, transmitted: 7
receive errors: 0, unknown types: 0
     runts: 0, aligns: 0, CRC: 0, parity: 0, overflow: 0
     too big: 0, out of buffers: 0, rcv timeout: 0, rcv reset: 0
transmit errors: 0
     collisions: 0, underflows: 0, timeouts: 0, resets: 0
```

```
    lost crs: 0, heartbeat failed: 0

ARP statistics:

arps received: 2 (0 requests, 2 replies)

    bad: opcodes: 0, hardware type: 0, protocol type: 0

arps transmitted: 4 (4 requests, 0 replies)

5 large buffers; 4 free now; minimum of 3 free

5 small buffers; 5 free now; minimum of 4 free
```

The message ping failed: Host unreachable for the last attempt is expected. PC/TCP provides the user with diagnostic messages with each ping command. To suppress these messages and simply get a success or fail message, the -z option can be used:

```
C:\> ping -z 147.120.0.11

host responding, time = 25 ms

C:\>

C:\> ping -z 147.120.0.1

host responding, time = 25 ms

C:\>

C:\> ping -z 147.120.0.10

ping failed: Host unreachable: ARP failed
```

If the ping command is not successful with the local address, either the network interface card is configured incorrectly or the software installation has incorrect parameters. Check the network card for the correct IRQ and memory settings and then check the cable to ensure that it is connected properly and network terminators are in

place. The software must have the correct drivers loaded, as well as the machine name, IP address, and similar information. If the local machine responds but the remote machines do not, check the network connections. Try ping from one of the remote machines to ensure that the DOS machine can be reached by the other machines.

Once the machines can successfully respond to a ping request, try ftp or telnet from the DOS-based machine. An ftp attempt to log onto the SCO UNIX machine is shown here:

```
FTP Software PC/TCP File Transfer Program 2.31    01/07/94 12:38

Copyright  1986-1993 by FTP Software, Inc.  All rights reserved.

FTP Trying....Open

220 tpci.tpci.com FTP Server (Version 5.60 #1) ready.

Userid for logging in on 147.120.0.1? tparker

331 Password required for tparker.

Password for logging in as tparker on 147.120.0.1? abcdefg

230 User tparker logged in.

ftp:147.120.0.1> ls

.profile

.lastlogin

.odtpref

trash

Initial.dt

XDesktop3

Transferred 265 bytes in 0 seconds
```

```
226 Transfer complete.

ftp:147.120.0.1> exit
```

This session, which displayed the listing of files on the SCO UNIX server, shows that the ftp command worked properly. The FTP session was closed with the command exit.

CHAPTER FOUR

WINSOCK AND THE SOCKET PROGRAMMING INTERFACE

4.1 Winsock

For some Windows and Windows 95 users, Winsock is the easiest method to get into TCP/IP because it is available from many public domain, BBS, and online service sites. There are several versions of Winsock, some of which are public domain or shareware. We will look at two versions of Winsock, one for Windows 3.X and another for Windows 95. We have chosen the popular Trumpet Winsock implementations for both operating systems because they are shareware, readily available, and well supported. Winsock is short for Windows Sockets, originally developed by Microsoft. Released in 1993, Windows Sockets is an interface for network programming in the Windows environment. Microsoft has published the specifications for Windows Sockets, hence making it an open application programming interface (API). The Winsock API (called WSA) is a library of function calls, data structures, and programming procedures that provide this standardized interface for applications. The second release of Winsock, called Winsock version 2, was released in mid 1995.

4.1.1 Trumpet Winsock

Trumpet Winsock is a shareware implementation of Winsock produced by Trumpet Software International. Trumpet Winsock is available for Windows 3.X and Windows 95 systems. Registration of the Winsock package, developed in Australia, is $25 US. Trumpet Winsock lets us use several different protocols including PPP and SLIP for connection to the Internet or remote networks, direct connection using TCP/IP, and the BOOTP protocol. Trumpet Winsock allows dynamic IP addressing, which is necessary with many Internet Service Providers. The Trumpet Winsock files are usually provided in an archive ZIP file, and should be extracted into a new subdirectory on our system. The primary files in the Trumpet Winsock distribution are

WINSOCK.DLL: The primary protocol stack for Winsock
TCPMAN.EXE: Manages the communications between WINSOCK.DLL and the network

TRUMPWSK.INI: Contains Winsock variable settings

HOSTS: A list of hosts that Winsock is aware of

SERVICES: A list of services supported by Winsock

PROTOCOL: A list of protocols supported by Winsock

There are a number of sample configuration files included in the archive, as well as utilities such as PING and HOP. Some of the files in the Winsock archive, such as HOSTS, PROTOCOL, and SERVICES, mirror UNIX files of the same name.

4.1.2 Installing Trumpet Winsock

The installation process for Trumpet Winsock is the same whether we are using SLIP/PPP for connection or a packet driver for LAN-based operations. We begin the installation by adding the directory holding the Trumpet Winsock files to our PATH. The files should, of course, be extracted from the ZIP file they are usually supplied in. After the path has been modified, we reboot our machine to effect the change.

We can create a Windows program group for the Trumpet Winsock system by adding a new program group from the Program Manager menus. (Select File menu, the New menu item, and then Program Group.) Create a title, such as Trumpet Winsock. for the new program group.

Next, we create a Program Icon for the TCPMAN program (the primary Trumpet Winsock program) by either creating a new Program Item from the Program Manager or opening the File Manager and dragging the TCPMAN.EXE entry from its directory to the Trumpet Winsock program group. Windows will prompt us for any information it needs. The program icon is read from the distribution files if the path is properly set.

To test the installation of the path and the Windows icon, we click the TCPMAN icon. If we receive error messages, either the PATH is not set properly or the program icon has not been properly defined. Because we are primarily interested in using Winsock on a TCP/IP network, we ignore configuring PPP and SLIP and concentrate on the TCP/IP stack.

4.1.3 Configuring the TCP/IP Packet Driver

Trumpet Winsock relies on a program called WINPKT to provide TCP/IP packet capabilities under Windows. After we create a program group for Winsock, we

need to set up the packet driver information in the network files. We will need a packet driver for our system, which is not included with most Trumpet Winsock distributions. In many cases, the network card vendor includes a disk with a packet driver on it. If not, one of the best sources for a packet driver is the Crynwr Packet Driver collection, a library of different packet drivers available from many online, BBS, FTP, and WWW sites. The packet driver specifications are added to our network startup batch file, usually AUTOEXEC.BAT for DOS-based systems.

The process for configuring Trumpet Winsock for LAN operation is quite simple. We set the IRQ and I/O address of the packet driver and add the packet driver to our system. A typical entry in the network batch file looks like this:

```
ne2000 0x60 2 0x300

WINPKT 0x60
```

This sets the network to use an NE2000 (Novell) type card, with I/O address of 300H, IRQ of 2, and a vector of 60. Several configurations are usually provided with the Trumpet Winsock distribution, although it is easy to derive our own from the network interface card manufacturer's documentation. To set up Trumpet Winsock for a packet driver, we use the Setup screen that appears when TCPMAN is first launched, or we use the menus within TCPMAN to display the setup screen. Deselect both Internal SLIP and Internal PPP settings. If either of them are checked, the packet driver will not launch properly. Enter the IP address, netmask, name server IP address, and domain name information. We may also modify the entries for Demand Load Time-out, MTU, TCP RWIN, TCP MSS, and TCP RTO MAX. The default values used for a packet driver are different than those for a SLIP/PPP setting. If we are using BOOTP or RARP to determine our machine IP address, we enter the proper protocol name in the IP address field.

The Packet Vector field should be set to the vector we used in the network card description, or we can leave it as 00 to let Trumpet Winsock search for the packet driver. After the configuration is saved, we restart TCPMAN and the network will be available (if the configuration and packet drivers are properly set). A ping command or similar utility will verify the packet driver operation is correct.

4.2 The Socket Programming Interface

Because the original socket interface was developed for UNIX systems, today's text has a decidedly UNIX-based orientation. However, the same principles apply to most other operating systems that support TCP/IP.

4.2.1 Development of the Socket Programming Interface

The basic structure of all socket programming commands lies with the unique structure of UNIX I/O. With UNIX, both input and output are treated as simple pipelines, where the input can be from anything and the output can go anywhere. The UNIX I/O system is sometimes referred to as the *open-read-write-close* system, because those are the steps that are performed for each I/O operation, whether it involves a file, a device, or a communications port. Whenever a file is involved, the UNIX operating system gives the file a *file descriptor*, a small number that uniquely identifies the file. A program can use this file descriptor to identify the file at any time. (The same holds true for a device; the process is the same.) A file operation uses an open function to return the file descriptor, which is used for the read (transfer data to the user's process) or write (transfer data from the user process to the file) functions, followed by a close function to terminate the file operation. The open function takes a filename as an argument. The read and write functions use the file descriptor number, the address of the buffer in which to read or write the information, and the number of bytes involved. The close function uses the file descriptor. The system is easy to use and simple to work with.

TCP/IP uses the same idea, relying on numbers to uniquely identify an end point for communications (a socket). Whenever the socket number is used, the operating system can resolve the socket number to the physical connector. An essential difference between a file descriptor and a socket number is that the socket requires some functions to be performed prior to the establishment of the socket (such as initialization). In techno-speak, "a file descriptor binds to a specific file or device when the open function is called, but the socket can be created without binding them to a specific destination at all (necessary for UDP), or bind them later (for TCP when the remote address is provided)." The same open-read-write-close procedure is used with sockets. The process was actually used literally with the first versions of TCP/IP. A special file called /dev/tcp was used as the device driver. The complexity added by networking made this approach awkward, though, so a library of special functions (the API) was developed.

The essential steps of open, read, write, and close are still followed in the protocol API.

4.2.2 Socket Services

There are three types of socket interfaces defined in the TCP/IP API. A socket can be used for TCP *stream communications,* in which a connection between two machines is created. It can be used for *UDP datagram communications,* a connectionless method of passing information between machines using packets of a predefined format. Or it can be used as a *raw* datagram process, in which the datagrams bypass the TCP/UDP layer and go straight to IP. The latter type arises from the fact that the socket API was not developed exclusively for TCP/IP.

The presence of all three types of interfaces can lead to problems with some parameters that depend exclusively on the type of interface. We must always bear in mind whether TCP or UDP is used. There are six basic communications commands that the socket API addresses through the TCP layer:

- open: Establishes a socket
- send: Sends data to the socket
- receive: Receives data from a socket
- status: Obtains status information about a socket
- close: Terminates a connection
- abort: Cancels an operation and terminates the connection

All six operations are logical and used as we would expect. The details for each step can be quite involved, but the basic operation remains the same. Many of the functions have been seen in previous days when dealing with specific protocols in some detail. Some of the functions (such as open) comprise several other functions that are available if necessary (such as establishing each end of the connection instead of both ends at once). Despite the formal definition of the functions within the API specifications, no formal method is given for how to implement them. There are two logical choices: synchronous, or *blocking,* in which the application waits for the command to complete before continuing execution; and asynchronous, or *nonblocking,* in which the application continues executing while the API function is processed. In the latter case, a function call further in the application's execution can check the API

functions' success and return codes. The problem with the synchronous or blocking method is that the application must wait for the function call to complete. If timeouts are involved, this can cause a noticeable delay for the user.

4.2.2.1 Transmission Control Block

The Transmission Control Block (TCB) is a complex data structure that contains details about a connection. The full TCB has over fifty fields in it. The existence of the TCB and the nature of the information it holds are key to the behavior of the socket interface.

4.2.2.2 Creating a Socket

The API lets a user create a socket whenever necessary with a simple function call. The function requires the family of the protocol to be used with the socket (so the operating system knows which type of socket to assign and how to decode information), the type of communication required, and the specific protocol. Such a function call is written as follows:

socket(*family*, *type*, *protocol*)

The *family* of the protocol actually specifies how the addresses are interpreted. Examples of families are TCP/IP (coded as AF_INET), Apple's AppleTalk (AF_APPLETALK), and UNIX filesystems (AF_UNIX). The exact protocol within the family is specified as the protocol parameter. When used, it specifically indicates the type of service that is to be used.

The *type* parameter indicates the type of communications used. It can be a connectionless datagram service (coded as SOCK_DGRAM), a stream delivery service (SOCK_STREAM), or a raw type (SOCK_RAW). The result from the function call is an integer that can be assigned to a variable for further checking.

4.2.2.3 Binding the Socket

Because a socket can be created without any binding to an address, there must be a function call to complete this process and establish the full connection. With the TCP/IP protocol, the socket function does not supply the local port number, the

destination port, or the IP address of the destination. The bind function is called to establish the local port address for the connection. Some applications (especially on a server) want to use a specific port for a connection. Other applications are content to let the protocol software assign a port. A specific port can be requested in the bind function. If it is available, the software allocates it and returns the port information. If the port cannot be allocated (it might be in use), a return code indicates an error in port assignment.

The bind function has the following format:

bind(*socket*, *local_address*, *address_length*)

socket is the integer number of the socket to which the bind is completed; *local_address* is the local address to which the bind is performed; and *address_length* is an integer that gives the length of the address in bytes. The address is not returned as a simple number but has the structure shown in Figure 4.1.

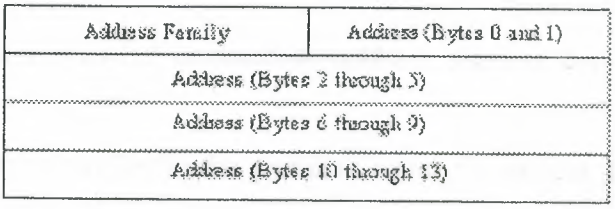

Figure 4.1 Address structure used by the socket API.

The address data structure (which is usually called sockaddr for *socket address*) has a 16-bit Address Family field that identifies the protocol family of the address. The entry in this field determines the format of the address in the following field (which might contain other information than the address, depending on how the protocol has defined the field). The Address field can be up to 14 bytes in length, although most protocols do not need this amount of space.

TCP/IP has a family address of 2, following which the Address field contains both a protocol port number (16 bits) and the IP address (32 bits). The remaining eight bytes are unused. This is shown in Figure 4.2. Because the address family defines how the Address field is decoded, there should be no problem with TCP/IP applications understanding the two pieces of information in the Address field.

Address Family (value = 2)	Protocol Port (16 bits)
IP Address (32 bits)	
Unused	
Unused	

Figure 4.2 The address structure for TCP/IP.

4.2.2.4 Connecting to the Destination

After a local socket address and port number have been assigned, the destination socket can be connected. A one-ended connection is referred to as being in an *unconnected state,* whereas a two-ended (complete) connection is in a *connected state.* After a bind function, an unconnected state exists. To become connected, the destination socket must be added to complete the connection.

To establish a connection to a remote socket, the connect function is used. The connect function's format is

connect(*socket, destination_address, address_length*)

The *socket* is the integer number of the socket to which to connect; the *destination_address* is the socket address data structure for the destination address (using the same format as shown in Figure 4.1); and the *address_length* is the length of the destination address in bytes.

The manner in which connect functions is protocol-dependent. For TCP, connect establishes the connection between the two endpoints and returns the information about the remote socket to the application. If a connection can't be established, an error message is generated. For a connectionless protocol such as UDP, the connect function is still necessary but stores only the destination address for the application.

4.2.2.5 The *open* Command

The open command prepares a communications port for communications. This is an alternative to the combination of the functions shown previously, used by applications for specific purposes. There are really three kinds of open commands, two

of which set a server to receive incoming requests and the third used by a client to initiate a request. With every open command, a TCB is created for that connection. The three open commands are an unspecified passive open (which enables a server to wait for a connection request from any client), a fully specified passive open (which enables a server to wait for a connection request from a specific client), and an active open (which initiates a connection with a server). The input and output expected from each command are shown in Table 4.1.

Table 4.1 Open command parameters.

Type	*Input*	*Output*
Unspecified	Local port	local connection name
passive open	Optional: timeout, precedence, security, maximum segment size	local connection name
Fully specified passive open	Local port, remote IP address, remote port Optional: timeout, precedence, security, maximum segment size	local connection name
Active open	Local port, destination IP address, destination port Optional: timeout, precedence, security, maximum segment size	local connection name

When an open command is issued by an application, a set of functions within the socket interface is executed to set up the TCB, initiate the socket number, and establish preliminary values for the variables used in the TCB and the application. The passive open command is issued by a server to wait for incoming requests. With the TCP (connection-based) protocol, the passive open issues the following function calls:

socket: Creates the sockets and identifies the type of communications.
bind: Establishes the server socket for the connection.

listen: Establishes a client queue.

accept: Waits for incoming connection requests on the socket.

The active open command is issued by a client. For TCP, it issues two functions:

socket: Creates the socket and identifies the communications type.

connect: Identifies the server's IP address and port; attempts to establish communications.

If the exact port to use is specified as part of the open command, a bind function call replaces the connect function.

4.2.2.6 Sending Data

There are five functions within the Socket API for sending data through a socket. These are send, sendto, sendmsg, write, and writev. Not surprisingly, all these functions send data from the application to TCP. They do this through a buffer created by the application (for example, it might be a memory address or a character string), passing the entire buffer to TCP. The send, write, and writev functions work only with a connected socket because they have no provision to specify a destination address within their function call. The format of the send function is simple. It takes the local socket connection number, the buffer address for the message to be sent, the length of the message in bytes, a Push flag, and an Urgent flag as parameters. An optional timeout might be specified. Nothing is returned as output from the send function. The format is

send(*socket, buffer_address, length, flags*)

The sendto and sendmsg functions are similar except they enable an application to send a message through an unconnected socket. They both require the destination address as part of their function call. The sendmsg function is simpler in format than the sendto function, primarily because another data structure is used to hold information. The sendmsg function is often used when the format of the sendto function would be awkward and inefficient in the application's code. Their formats are

sendto(*socket, buffer_address, length, flags, destination, address_length*)

sendmsg(*socket, message_structure, flags*)

The last two parameters in the sendto function are the destination address and the length of the destination address. The address is specified using the format shown in Figure 4.1. The *message_structure* of the sendmsg function contains the information left out of the sendto function call. The format of the message structure is shown in Figure 4.3.

Pointer to Socket Address
Size of Socket Address (in bytes)
Pointer to iovector List (Message)
Length of iovector list
Destination Address
Length of Desination Address

Figure 4.3 The message structure used by sendmsg.

The fields in the sendmsg message structure give the socket address, size of the socket address, a pointer to the iovector, which contains information about the message to be sent, the length of the iovector, the destination address, and the length of the destination address. The iovector is an address for an array that points to the message to be sent. The array is a set of pointers to the bytes that comprise the message. The format of the iovector is simple. For each 32-bit address to a memory location with a chunk of the message, a corresponding 32-bit field holds the length of the message in that memory location. This format is repeated until the entire message is specified. This is shown in Figure 4.4. The iovector format enables a noncontiguous message to be sent. In other words, the first part of the message can be in one location in memory, and the rest is separated by other information. This can be useful because it saves the application from copying long messages into a contiguous location.

Pointer to Message Block 1 (32 bits)
Length of Message in Block 1 (32 bits)
Pointer to Message Block 2 (32 bits)
Length of Message in Block 2 (32 bits)
...
Pointer to Message in Block n (32 bits)
Length of Message in Block n (32 bits)

Figure 4.4 The iovector format.

The write function takes three arguments: the socket number, the buffer address of the message to be sent, and the length of the message to send. The format of the function call is

write(*socket*, *buffer_address*, *length*)

The writev function is similar to write except it uses the iovector to hold the message. This lets it send a message without copying it into another memory address. The format of writev is

writev(*socket*, *iovector*, *length*)

where *length* is the number of entries in iovector.

The type of function chosen to send data through a socket depends on the type of connection used and the level of complexity of the application. To a considerable degree, it is also a personal choice of the programmer.

4.2.2.7 Receiving Data

Not surprisingly, because there are five functions to send data through a socket, there are five corresponding functions to receive data: read, readv, recv, recvfrom, and recvmsg. They all accept incoming data from a socket into a reception buffer. The receive buffer can then be transferred from TCP to the application.
The read function is the simplest and can be used only when a socket is connected. Its format is

read(*socket*, *buffer*, *length*)

The first parameter is the number of the socket or a file descriptor from which to read the data, followed by the memory address in which to store the incoming data, and the maximum number of bytes to be read.

As with writev, the readv command enables incoming messages to be placed in noncontiguous memory locations through the use of an iovector. The format of readv is

readv(*socket*, *iovector*, *length*)

length is the number of entries in the iovector. The format of the iovector is the same as mentioned previously and shown in Figure 4.4.

The recv function also can be used with connected sockets. It has the format

recv(*socket, buffer_address, length, flags*)

which corresponds to the send function's arguments.

The recvfrom and recvmsg functions enable data to be read from an unconnected socket. Their formats include the sender's address:

recvfrom(*socket, buffer_address, length, flags, source_address, address_length*)

recvmsg(*socket, message_structure, flags*)

The message structure in the recvmsg function corresponds to the structure in sendmsg. (Figure 4.3.)

4.2.2.8 Server Listening

A server application that expects clients to call in to it has to create a socket (using socket), bind it to a port (with bind), then wait for incoming requests for data. The listen function handles problems that could occur with this type of behavior by establishing a queue for incoming connection requests. The queue prevents bottlenecks and collisions, such as when a new request arrives before a previous one has been completely handled, or two requests arrive simultaneously. The listen function establishes a buffer to queue incoming requests, thereby avoiding losses. The function lets the socket accept incoming connection requests, which are all sent to the queue for future processing. The function's format is

listen(*socket, queue_length*)

where *queue_length* is the size of the incoming buffer. If the buffer has room, incoming requests for connections are added to the buffer and the application can deal with them in the order of reception. If the buffer is full, the connection request is rejected.

After the server has used listen to set up the incoming connection request queue, the accept function is used to actually wait for a connection. The format of the function is

accept(*socket*, *address*, *length*)

socket is the socket on which to accept requests; *address* is a pointer to a structure similar to Figure 4.1; and *length* is a pointer to an integer showing the length of the address.

When a connection request is received, the protocol places the address of the client in the memory location indicated by the address parameter, and the length of that address in the length location. It then creates a new socket that has the client and server connected together, sending back the socket description to the client. The socket on which the request was received remains open for other connection requests. This enables multiple requests for a connection to be processed, whereas if that socket was closed down with each connection request, only one client/server process could be handled at a time.

One possible special occurrence must be handled on UNIX systems. It is possible for a single process to wait for a connection request on multiple sockets. This reduces the number of processes that monitor sockets, thereby lowering the amount of overhead the machine uses. To provide for this type of process, the select function is used. The format of the function is

select(*num_desc*, *in_desc*, *out_desc*, *excep_desc*, *timeout*)

num_desc is the number of sockets or descriptors that are monitored; *in_desc* and *out_desc* are pointers to a bit mask that indicates the sockets or file descriptors to monitor for input and output, respectively; *excep_desc* is a pointer to a bit mask that specifies the sockets or file descriptors to check for exception conditions; and *timeout* is a pointer to an integer that indicates how long to wait (a value of 0 indicates forever). To use the select function, a server creates all the necessary sockets first, then calls select to determine which ones are for input, output, and exceptions.

4.2.2.9 Getting Status Information

Several status functions are used to obtain information about a connection. They can be used at any time, although they are typically used to establish the integrity of a connection in case of problems or to control the behavior of the socket. The status functions require the name of the local connection, and they return a set of information, which might include the local and remote socket names, local connection name, receive and send window states, number of buffers waiting for an acknowledgment, number of buffers waiting for data, and current values for the urgent state, precedence, security, and timeout variables. Most of this information is read from the Transmission Control Block (TCB). The format of the information and the exact contents vary slightly, depending on the implementation.

The function getsockopt enables an application to query the socket for information. The function format is

getsockopt(*socket*, *level*, *option_id*, *option_result*, *length*)

socket is the number of the socket; *level* indicates whether the function refers to the socket itself or the protocol that uses it; *option_id* is a single integer that identifies the type of information requested; *option_result* is a pointer to a memory location where the function should place the result of the query; and *length* is the length of the result.
The corresponding setsockopt function lets the application set a value for the socket. The function's format is the same as getsockopt except that *option_result* points to the value that is to be set, and *length* is the length of the value.
Two functions provide information about the local address of a socket. The getpeername function returns the address of the remote end. The getsockname function returns the local address of a socket. They have the following formats:

getpeername(*socket*, *destination_address*, *address_length*)

getsockname(*socket*, *local_address*, *address_length*)

The addresses in both functions are pointers to a structure of the format shown in Figure 4.1. Two host name functions for BSD UNIX are gethostname and sethostname, which

enable an application to obtain the name of the host and set the host name (if permissions allow). Their formats are as follows:

sethostname(*name*, *length*)

gethostname(*name*, *length*)

The *name* is the address of an array that holds the name, and the *length* is an integer that gives the name's length.

A similar set of functions provides for domain names. The functions setdomainname and getdomainname enable an application to obtain or set the domain names. Their formats are

setdomainname(*name*, *length*)

getdomainname(*name*, *length*)

The parameters are the same as with the sethostname and gethostname functions, except for the format of the name (which reflects domain name format).

4.2.2.10 Closing a Connection

The close function closes a connection. It requires only the local connection name to complete the process. It also takes care of the TCB and releases any variable created by the connection. No output is generated.

The close function is initiated with the call

close(*socket*)

where the socket name is required. If an application terminates abnormally, the operating system closes all sockets that were open prior to the termination.

4.2.2.11 Aborting a Connection

The abort function instructs TCP to discard all data that currently resides in send and receive buffers and close the connection. It takes the local connection name as input. No output is generated. This function can be used in case of emergency shutdown routines, or in case of a fatal failure of the connection or associated software.
The abort function is usually implemented by the close() call, although some special instructions might be available with different implementations.

4.2.2.12 UNIX Forks

UNIX has two system calls that can affect sockets: fork and exec. Both are frequently used by UNIX developers because of their power. (In fact, forks are one of the most powerful tools UNIX offers, and one that most other operating systems lack.) For simplicity, we deal with the two functions as though they perform the same task.
A fork call creates a copy of the existing application as a new process and starts executing it. The new process has all the original's file descriptors and socket information. This can cause a problem if the application programmer didn't take into account the fact that two (or more) processes try to use the same socket (or file) simultaneously. Therefore, applications that can fork have to take into account potential conflicts and code around them by checking the status of shared sockets.

The operating system itself keeps a table of each socket and how many processes have access to it. An internal counter is incremented or decremented with each process's open or close function call for the socket. When the last process using a socket is terminated, the socket is permanently closed. This prevents one forked process from closing a socket when its original is still using it.

CONCLUSION

TCP/UDP are the standard, routable entries networking protocols. All modern operating systems offer TCP support and most large networks rely on TCP for much of their network traffic. This is a technology for connecting dissimilar systems. Many standard connectivity utilities are available to access and transfer data between dissimilar systems, including File Transfer Protocol and Telnet. It provides a robust, scalable, cross-platform client/server framework. TCP/IP offers the socket interface, which is ideal for developing client/server applications that can run on Sockets-compliant stacks from other venders. Sockets applications can also advantage of other networking protocols such as NWLink used in Novell Net Ware networks. TCP/IP provides a method of gaining access to the Internet. The internet consists of thousands of network wordwide connecting research facilities, universities, libraries, government agencies and private companies.

REFERENCES

[1] James Chellis, Charles Perkings, Matthew Strebe, "Networking Essentials", SYBEX Publishers 1999.

[2] James Chellis, "Windows 2000 Network Infra Structure" SYBEX Publishers 2000

[3] Charles W. "Understanding TCP/IP" BPB Publishers 1996

http://www.us-epanorama.net

http://www.microsoft.com

http://www.commweb.com

http://www.oreily.com