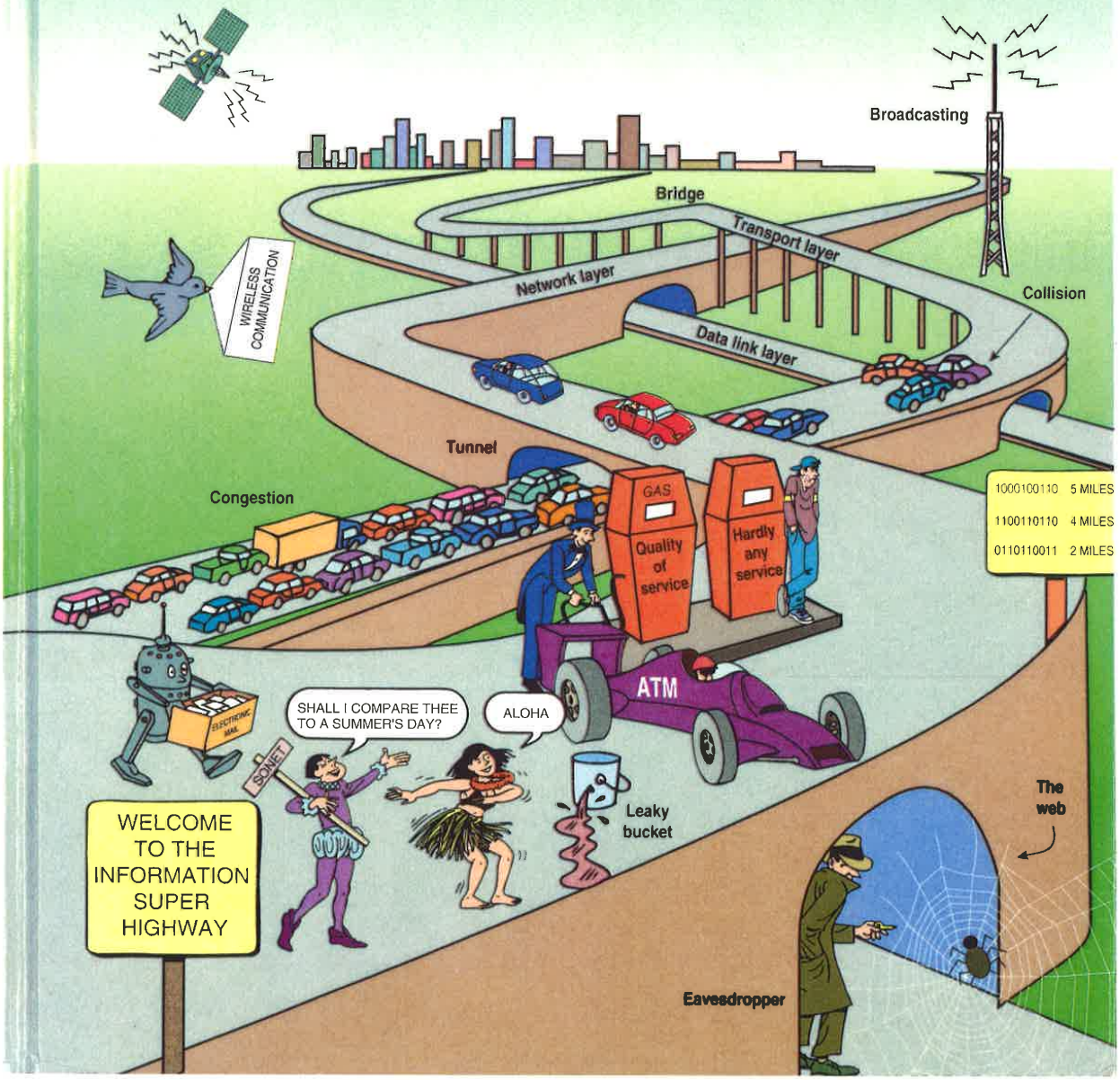
FINNEGAN PAO LIBRARY
PAO 003542

THIRD EDITION

COMPUTER NETWORKS

ANDREW S. TANENBAUM



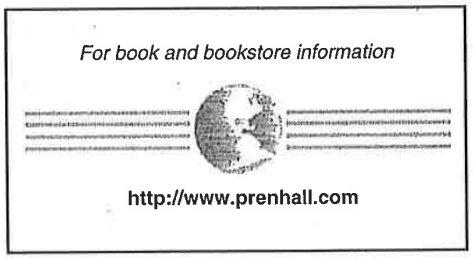
OpenTV Exhibit 2004
Apple v. OpenTV
IPR2015-00969

Computer Networks

Third Edition

Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands



Prentice Hall PTR
Upper Saddle River, New Jersey 07458

Library of Congress Cataloging in Publication Data

Tanenbaum, Andrew S. 1944-.
Computer networks / Andrew S. Tanenbaum. -- 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-349945-6
1.Computer networks. I. Title.
TK5105.5.T36 1996 96-4121
004.6--dc20 CIP

Editorial/production manager: *Camille Trentacoste*
Interior design and composition: *Andrew S. Tanenbaum*
Cover design director: *Jerry Votta*
Cover designer: *Don Martinetti, DM Graphics, Inc.*
Cover concept: *Andrew S. Tanenbaum, from an idea by Marilyn Tremaine*
Interior graphics: *Hadel Studio*
Manufacturing manager: *Alexis R. Heydt*
Acquisitions editor: *Mary Franz*
Editorial Assistant: *Noreen Regina*

© 1996 by Prentice Hall PTR
Prentice-Hall, Inc.
A Simon & Schuster Company
Upper Saddle River, New Jersey 07458

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:
Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458.
Phone: (800) 382-3419; Fax: (201) 236-7141. E-mail: corpsales@prenhall.com

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

All product names mentioned herein are the trademarks of their respective owners.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-349945-6

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

1

INTRODUCTION

Each of the past three centuries has been dominated by a single technology. The 18th Century was the time of the great mechanical systems accompanying the Industrial Revolution. The 19th Century was the age of the steam engine. During the 20th Century, the key technology has been information gathering, processing, and distribution. Among other developments, we have seen the installation of worldwide telephone networks, the invention of radio and television, the birth and unprecedented growth of the computer industry, and the launching of communication satellites.

Due to rapid technological progress, these areas are rapidly converging, and the differences between collecting, transporting, storing, and processing information are quickly disappearing. Organizations with hundreds of offices spread over a wide geographical area routinely expect to be able to examine the current status of even their most remote outpost at the push of a button. As our ability to gather, process, and distribute information grows, the demand for even more sophisticated information processing grows even faster.

Although the computer industry is young compared to other industries (e.g., automobiles and air transportation), computers have made spectacular progress in a short time. During the first two decades of their existence, computer systems were highly centralized, usually within a single large room. Not infrequently, this room had glass walls, through which visitors could gawk at the great electronic wonder inside. A medium-size company or university might have had one or two

computers, while large institutions had at most a few dozen. The idea that within 20 years equally powerful computers smaller than postage stamps would be mass produced by the millions was pure science fiction.

The merging of computers and communications has had a profound influence on the way computer systems are organized. The concept of the "computer center" as a room with a large computer to which users bring their work for processing is now totally obsolete. The old model of a single computer serving all of the organization's computational needs has been replaced by one in which a large number of separate but interconnected computers do the job. These systems are called **computer networks**. The design and organization of these networks are the subjects of this book.

Throughout the book we will use the term "computer network" to mean an *interconnected* collection of *autonomous* computers. Two computers are said to be interconnected if they are able to exchange information. The connection need not be via a copper wire; fiber optics, microwaves, and communication satellites can also be used. By requiring the computers to be autonomous, we wish to exclude from our definition systems in which there is a clear master/slave relation. If one computer can forcibly start, stop, or control another one, the computers are not autonomous. A system with one control unit and many slaves is not a network; nor is a large computer with remote printers and terminals.

There is considerable confusion in the literature between a computer network and a **distributed system**. The key distinction is that in a distributed system, the existence of multiple autonomous computers is transparent (i.e., not visible) to the user. He† can type a command to run a program, and it runs. It is up to the operating system to select the best processor, find and transport all the input files to that processor, and put the results in the appropriate place.

In other words, the user of a distributed system is not aware that there are multiple processors; it looks like a virtual uniprocessor. Allocation of jobs to processors and files to disks, movement of files between where they are stored and where they are needed, and all other system functions must be automatic.

With a network, users must *explicitly* log onto one machine, *explicitly* submit jobs remotely, *explicitly* move files around and generally handle all the network management personally. With a distributed system, nothing has to be done explicitly; it is all automatically done by the system without the users' knowledge.

In effect, a distributed system is a software system built on top of a network. The software gives it a high degree of cohesiveness and transparency. Thus the distinction between a network and a distributed system lies with the software (especially the operating system), rather than with the hardware.

Nevertheless, there is considerable overlap between the two subjects. For example, both distributed systems and computer networks need to move files around. The difference lies in who invokes the movement, the system or the user.

† "He" should be read as "he or she" throughout this book.

Although this book primarily focuses on networks, many of the topics are also important in distributed systems. For more information about distributed systems, see (Coulouris et al., 1994; Mullender, 1993; and Tanenbaum, 1995).

1.1. USES OF COMPUTER NETWORKS

Before we start to examine the technical issues in detail, it is worth devoting some time to pointing out why people are interested in computer networks and what they can be used for.

1.1.1. Networks for Companies

Many organizations have a substantial number of computers in operation, often located far apart. For example, a company with many factories may have a computer at each location to keep track of inventories, monitor productivity, and do the local payroll. Initially, each of these computers may have worked in isolation from the others, but at some point, management may have decided to connect them to be able to extract and correlate information about the entire company.

Put in slightly more general form, the issue here is **resource sharing**, and the goal is to make all programs, equipment, and especially data available to anyone on the network without regard to the physical location of the resource and the user. In other words, the mere fact that a user happens to be 1000 km away from his data should not prevent him from using the data as though they were local. This goal may be summarized by saying that it is an attempt to end the "tyranny of geography."

A second goal is to provide **high reliability** by having alternative sources of supply. For example, all files could be replicated on two or three machines, so if one of them is unavailable (due to a hardware failure), the other copies could be used. In addition, the presence of multiple CPUs means that if one goes down, the others may be able to take over its work, although at reduced performance. For military, banking, air traffic control, nuclear reactor safety, and many other applications, the ability to continue operating in the face of hardware problems is of utmost importance.

Another goal is **saving money**. Small computers have a much better price/performance ratio than large ones. Mainframes (room-size computers) are roughly a factor of ten faster than personal computers, but they cost a thousand times more. This imbalance has caused many systems designers to build systems consisting of personal computers, one per user, with data kept on one or more shared **file server** machines. In this model, the users are called **clients**, and the whole arrangement is called the **client-server model**. It is illustrated in Fig. 1-1.

In the client-server model, communication generally takes the form of a request message from the client to the server asking for some work to be done.

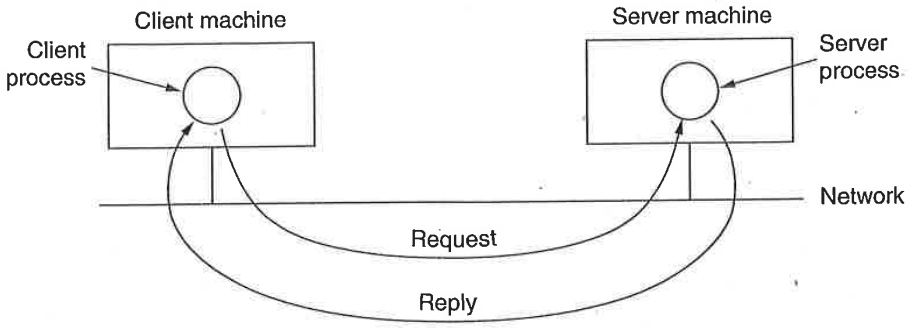


Fig. 1-1. The client-server model.

The server then does the work and sends back the reply. Usually, there are many clients using a small number of servers.

Another networking goal is scalability, the ability to increase system performance gradually as the workload grows just by adding more processors. With centralized mainframes, when the system is full, it must be replaced by a larger one, usually at great expense and even greater disruption to the users. With the client-server model, new clients and new servers can be added as needed.

Yet another goal of setting up a computer network has little to do with technology at all. A computer network can provide a powerful **communication medium** among widely separated employees. Using a network, it is easy for two or more people who live far apart to write a report together. When one worker makes a change to an on-line document, the others can see the change immediately, instead of waiting several days for a letter. Such a speedup makes cooperation among far-flung groups of people easy where it previously had been impossible. In the long run, the use of networks to enhance human-to-human communication will probably prove more important than technical goals such as improved reliability.

1.1.2. Networks for People

The motivations given above for building computer networks are all essentially economic and technological in nature. If sufficiently large and powerful mainframes were available at acceptable prices, most companies would simply choose to keep all their data on them and give employees terminals connected to them. In the 1970s and early 1980s, most companies operated this way. Computer networks only became popular when networks of personal computers offered a huge price/performance advantage over mainframes.

Starting in the 1990s, computer networks began to start delivering services to private individuals at home. These services and the motivations for using them

are quite different than the "corporate efficiency" model described in the previous section. Below we will sketch three of the more exciting ones that are starting to happen:

1. Access to remote information.
2. Person-to-person communication.
3. Interactive entertainment.

Access to remote information will come in many forms. One area in which it is already happening is access to financial institutions. Many people pay their bills, manage their bank accounts, and handle their investments electronically. Home shopping is also becoming popular, with the ability to inspect the on-line catalogs of thousands of companies. Some of these catalogs will soon provide the ability to get an instant video on any product by just clicking on the product's name.

Newspapers will go on-line and be personalized. It will be possible to tell the newspaper that you want everything about corrupt politicians, big fires, scandals involving celebrities, and epidemics, but no football, thank you. At night while you sleep, the newspaper will be downloaded to your computer's disk or printed on your laser printer. On a small scale, this service already exists. The next step beyond newspapers (plus magazines and scientific journals) is the on-line digital library. Depending on the cost, size, and weight of book-sized notebook computers, printed books may become obsolete. Skeptics should take note of the effect the printing press had on the medieval illuminated manuscript.

Another application that falls in this category is access to information systems like the current World Wide Web, which contains information about the arts, business, cooking, government, health, history, hobbies, recreation, science, sports, travel, and too many other topics to even mention.

All of the above applications involve interactions between a person and a remote database. The second broad category of network use will be person-to-person interactions, basically the 21st Century's answer to the 19th Century's telephone. Electronic mail or **email** is already widely used by millions of people and will soon routinely contain audio and video as well as text. Smell in messages will take a bit longer to perfect.

Real-time email will allow remote users to communicate with no delay, possibly seeing and hearing each other as well. This technology makes it possible to have virtual meetings, called **videoconference**, among far-flung people. It is sometimes said that transportation and communication are having a race, and whichever wins will make the other obsolete. Virtual meetings could be used for remote school, getting medical opinions from distant specialists, and numerous other applications.

Worldwide newsgroups, with discussions on every conceivable topic are already commonplace among a select group of people, and this will grow to

include the population at large. These discussions, in which one person posts a message and all the other subscribers to the newsgroup can read it, run the gamut from humorous to impassioned.

Our third category is entertainment, which is a huge and growing industry. The killer application here (the one that may drive all the rest) is video on demand. A decade or so hence, it may be possible to select any movie or television program ever made, in any country, and have it displayed on your screen instantly. New films may become interactive, where the user is occasionally prompted for the story direction (should MacBeth murder Duncan or just bide his time?) with alternative scenarios provided for all cases. Live television may also become interactive, with the audience participating in quiz shows, choosing among contestants, and so on.

On the other hand, maybe the killer application will not be video on demand. Maybe it will be game playing. Already we have multiperson real-time simulation games, like hide-and-seek in a virtual dungeon, and flight simulators with the players on one team trying to shoot down the players on the opposing team. If done with goggles and 3-dimensional real-time, photographic-quality moving images, we have a kind of worldwide shared virtual reality.

In short, the ability to merge information, communication, and entertainment will surely give rise to a massive new industry based on computer networking.

1.1.3. Social Issues

The widespread introduction of networking will introduce new social, ethical, political problems (Laudon, 1995). Let us just briefly mention a few of them; a thorough study would require a full book, at least. A popular feature of many networks are newsgroups or bulletin boards where people can exchange messages with like-minded individuals. As long as the subjects are restricted to technical topics or hobbies like gardening, not too many problems will arise.

The trouble comes when newsgroups are set up on topics that people actually care about, like politics, religion, or sex. Views posted to such groups may be deeply offensive to some people. Furthermore, messages need not be limited to text. High-resolution color photographs and even short video clips can now easily be transmitted over computer networks. Some people take a live-and-let-live view, but others feel that posting certain material (e.g., child pornography) is simply unacceptable. Thus the debate rages.

People have sued network operators, claiming that they are responsible for the contents of what they carry, just as newspapers and magazines are. The inevitable response is that a network is like a telephone company or the post office and cannot be expected to police what its users say. Stronger yet, having network operators censor messages would probably cause them to delete everything with even the slightest possibility of their being sued, and thus violate their users' rights to free speech. It is probably safe to say that this debate will go on for a while.

Another fun area is employee rights versus employer rights. Many people read and write email at work. Some employers have claimed the right to read and possibly censor employee messages, including messages sent from a home terminal after work. Not all employees agree with this (Sipior and Ward, 1995).

Even if employers have power over employees, does this relationship also govern universities and students? How about high schools and students? In 1994, Carnegie-Mellon University decided to turn off the incoming message stream for several newsgroups dealing with sex because the university felt the material was inappropriate for minors (i.e., those few students under 18). The fallout from this event will take years to settle.

Computer networks offer the potential for sending anonymous messages. In some situations, this capability may be desirable. For example, it provides a way for students, soldiers, employees, and citizens to blow the whistle on illegal behavior on the part of professors, officers, superiors, and politicians without fear of reprisals. On the other hand, in the United States and most other democracies, the law specifically permits an accused person the right to confront and challenge his accuser in court. Anonymous accusations cannot be used as evidence.

In short, computer networks, like the printing press 500 years ago, allow ordinary citizens to distribute their views in different ways and to different audiences than were previously possible. This new-found freedom brings with it many unsolved social, political, and moral issues. The solution to these problems is left as an exercise for the reader.

1.2. NETWORK HARDWARE

It is now time to turn our attention from the applications and social aspects of networking to the technical issues involved in network design. There is no generally accepted taxonomy into which all computer networks fit, but two dimensions stand out as important: transmission technology and scale. We will now examine each of these in turn.

Broadly speaking, there are two types of transmission technology:

1. Broadcast networks.
2. Point-to-point networks.

Broadcast networks have a single communication channel that is shared by all the machines on the network. Short messages, called **packets** in certain contexts, sent by any machine are received by all the others. An address field within the packet specifies for whom it is intended. Upon receiving a packet, a machine checks the address field. If the packet is intended for itself, it processes the packet; if the packet is intended for some other machine, it is just ignored.

As an analogy, consider someone standing at the end of a corridor with many rooms off it and shouting "Watson, come here. I want you." Although the packet

may actually be received (heard) by many people, only Watson responds. The others just ignore it. Another example is an airport announcement asking all flight 644 passengers to report to gate 12.

Broadcast systems generally also allow the possibility of addressing a packet to *all* destinations by using a special code in the address field. When a packet with this code is transmitted, it is received and processed by every machine on the network. This mode of operation is called **broadcasting**. Some broadcast systems also support transmission to a subset of the machines, something known as **multicasting**. One possible scheme is to reserve one bit to indicate multicasting. The remaining $n - 1$ address bits can hold a group number. Each machine can "subscribe" to any or all of the groups. When a packet is sent to a certain group, it is delivered to all machines subscribing to that group.

In contrast, **point-to-point** networks consist of many connections between individual pairs of machines. To go from the source to the destination, a packet on this type of network may have to first visit one or more intermediate machines. Often multiple routes, of different lengths are possible, so routing algorithms play an important role in point-to-point networks. As a general rule (although there are many exceptions), smaller, geographically localized networks tend to use broadcasting, whereas larger networks usually are point-to-point.

Interprocessor distance	Processors located in same	Example
0.1 m	Circuit board	Data flow machine
1 m	System	Multicomputer
10 m	Room	Local area network
100 m	Building	
1 km	Campus	
10 km	City	Metropolitan area network
100 km	Country	Wide area network
1,000 km	Continent	
10,000 km	Planet	The internet

Fig. 1-2. Classification of interconnected processors by scale.

An alternative criterion for classifying networks is their scale. In Fig. 1-2 we give a classification of multiple processor systems arranged by their physical size. At the top are **data flow machines**, highly parallel computers with many functional units all working on the same program. Next come the **multicomputers**, systems that communicate by sending messages over very short, very fast buses. Beyond the multicomputers are the true networks, computers that communicate

by exchanging messages over longer cables. These can be divided into local, metropolitan, and wide area networks. Finally, the connection of two or more networks is called an internetwork. The worldwide Internet is a well-known example of an internetwork. Distance is important as a classification metric because different techniques are used at different scales. In this book we will be concerned with only the true networks and their interconnection. Below we give a brief introduction to the subject of network hardware.

1.2.1. Local Area Networks

Local area networks, generally called **LANs**, are privately-owned networks within a single building or campus of up to a few kilometers in size. They are widely used to connect personal computers and workstations in company offices and factories to share resources (e.g., printers) and exchange information. LANs are distinguished from other kinds of networks by three characteristics: (1) their size, (2) their transmission technology, and (3) their topology.

LANs are restricted in size, which means that the worst-case transmission time is bounded and known in advance. Knowing this bound makes it possible to use certain kinds of designs that would not otherwise be possible. It also simplifies network management.

LANs often use a transmission technology consisting of a single cable to which all the machines are attached, like the telephone company party lines once used in rural areas. Traditional LANs run at speeds of 10 to 100 Mbps, have low delay (tens of microseconds), and make very few errors. Newer LANs may operate at higher speeds, up to hundreds of megabits/sec. In this book, we will adhere to tradition and measure line speeds in megabits/sec (Mbps), not megabytes/sec (MB/sec). A megabit is 1,000,000 bits, not 1,048,576 (2^{20}) bits.

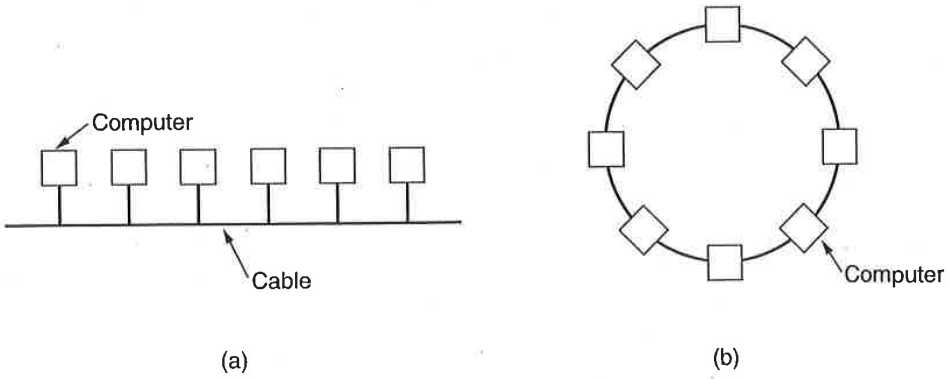


Fig. 1-3. Two broadcast networks. (a) Bus. (b) Ring.

Various topologies are possible for broadcast LANs. Figure 1-3 shows two of them. In a bus (i.e., a linear cable) network, at any instant one machine is the

master and is allowed to transmit. All other machines are required to refrain from sending. An arbitration mechanism is needed to resolve conflicts when two or more machines want to transmit simultaneously. The arbitration mechanism may be centralized or distributed. IEEE 802.3, popularly called **Ethernet**™, for example, is a bus-based broadcast network with decentralized control operating at 10 or 100 Mbps. Computers on an Ethernet can transmit whenever they want to; if two or more packets collide, each computer just waits a random time and tries again later.

A second type of broadcast system is the ring. In a ring, each bit propagates around on its own, not waiting for the rest of the packet to which it belongs. Typically, each bit circumnavigates the entire ring in the time it takes to transmit a few bits, often before the complete packet has even been transmitted. Like all other broadcast systems, some rule is needed for arbitrating simultaneous accesses to the ring. Various methods are in use and will be discussed later in this book. IEEE 802.5 (the IBM token ring), is a popular ring-based LAN operating at 4 and 16 Mbps.

Broadcast networks can be further divided into static and dynamic, depending on how the channel is allocated. A typical static allocation would be to divide up time into discrete intervals and run a round robin algorithm, allowing each machine to broadcast only when its time slot comes up. Static allocation wastes channel capacity when a machine has nothing to say during its allocated slot, so most systems attempt to allocate the channel dynamically (i.e., on demand).

Dynamic allocation methods for a common channel are either centralized or decentralized. In the centralized channel allocation method, there is a single entity, for example a bus arbitration unit, which determines who goes next. It might do this by accepting requests and making a decision according to some internal algorithm. In the decentralized channel allocation method, there is no central entity; each machine must decide for itself whether or not to transmit. You might think that this always leads to chaos, but it does not. Later we will study many algorithms designed to bring order out of the potential chaos.

The other kind of LAN is built using point-to-point lines. Individual lines connect a specific machine with another specific machine. Such a LAN is really a miniature wide area network. We will look at these later.

1.2.2. Metropolitan Area Networks

A **metropolitan area network**, or **MAN** (plural: MANs, not MEN) is basically a bigger version of a LAN and normally uses similar technology. It might cover a group of nearby corporate offices or a city and might be either private or public. A MAN can support both data and voice, and might even be related to the local cable television network. A MAN just has one or two cables and does not contain switching elements, which shunt packets over one of several potential output lines. Not having to switch simplifies the design.

The main reason for even distinguishing MANs as a special category is that a standard has been adopted for them, and this standard is now being implemented. It is called **DQDB (Distributed Queue Dual Bus)** or for people who prefer numbers to letters, 802.6 (the number of the IEEE standard that defines it). DQDB consists of two unidirectional buses (cables) to which all the computers are connected, as shown in Fig. 1-4. Each bus has a head-end, a device that initiates transmission activity. Traffic that is destined for a computer to the right of the sender uses the upper bus. Traffic to the left uses the lower one.

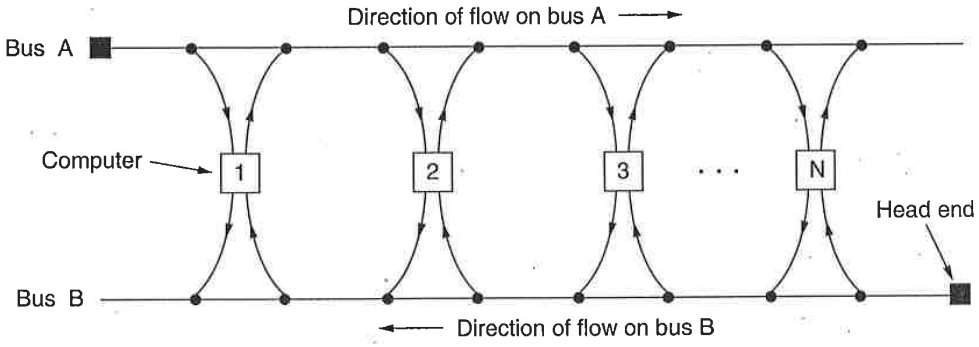


Fig. 1-4. Architecture of the DQDB metropolitan area network.

A key aspect of a MAN is that there is a broadcast medium (for 802.6, two cables) to which all the computers are attached. This greatly simplifies the design compared to other kinds of networks. We will discuss DQDB in more detail in Chap. 4.

1.2.3. Wide Area Networks

A **wide area network**, or **WAN**, spans a large geographical area, often a country or continent. It contains a collection of machines intended for running user (i.e., application) programs. We will follow traditional usage and call these machines **hosts**. The term **end system** is sometimes also used in the literature. The hosts are connected by a **communication subnet**, or just **subnet** for short. The job of the subnet is to carry messages from host to host, just as the telephone system carries words from speaker to listener. By separating the pure communication aspects of the network (the subnet) from the application aspects (the hosts), the complete network design is greatly simplified.

In most wide area networks, the subnet consists of two distinct components: transmission lines and switching elements. Transmission lines (also called **circuits**, **channels**, or **trunks**) move bits between machines.

The switching elements are specialized computers used to connect two or more transmission lines. When data arrive on an incoming line, the switching

element must choose an outgoing line to forward them on. Unfortunately, there is no standard terminology used to name these computers. They are variously called **packet switching nodes**, **intermediate systems**, and **data switching exchanges**, among other things. As a generic term for the switching computers, we will use the word **router**, but the reader should be aware that no consensus on terminology exists here. In this model, shown in Fig. 1-5, each host is generally connected to a LAN on which a router is present, although in some cases a host can be connected directly to a router. The collection of communication lines and routers (but not the hosts) form the subnet.

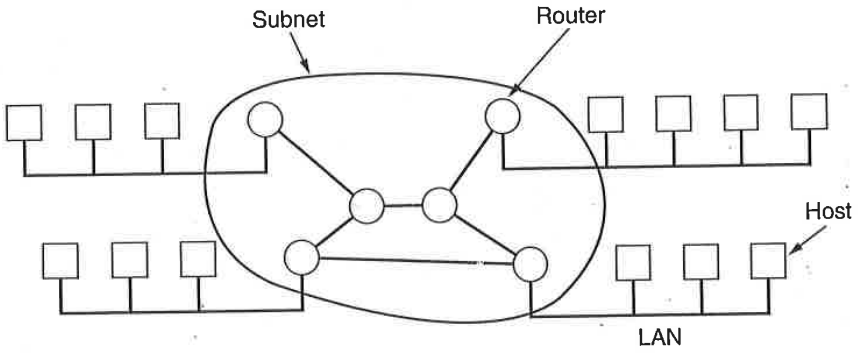


Fig. 1-5. Relation between hosts and the subnet.

An aside about the term "subnet" is worth making. Originally, its only meaning was the collection of routers and communication lines that moved packets from the source host to the destination host. However, some years later, it also acquired a second meaning in conjunction with network addressing (which we will discuss in Chap. 5). Hence the term has a certain ambiguity about it. Unfortunately, no widely-used alternative exists for its initial meaning, so with some hesitation we will use it in both senses. From the context, it will always be clear which is meant.

In most WANs, the network contains numerous cables or telephone lines, each one connecting a pair of routers. If two routers that do not share a cable nevertheless wish to communicate, they must do this indirectly, via other routers. When a packet is sent from one router to another via one or more intermediate routers, the packet is received at each intermediate router in its entirety, stored there until the required output line is free, and then forwarded. A subnet using this principle is called a **point-to-point**, **store-and-forward**, or **packet-switched** subnet. Nearly all wide area networks (except those using satellites) have store-and-forward subnets. When the packets are small and all the same size, they are often called **cells**.

When a point-to-point subnet is used, an important design issue is what the router interconnection topology should look like. Figure 1-6 shows several

possible topologies. Local networks that were designed as such usually have a symmetric topology. In contrast, wide area networks typically have irregular topologies.

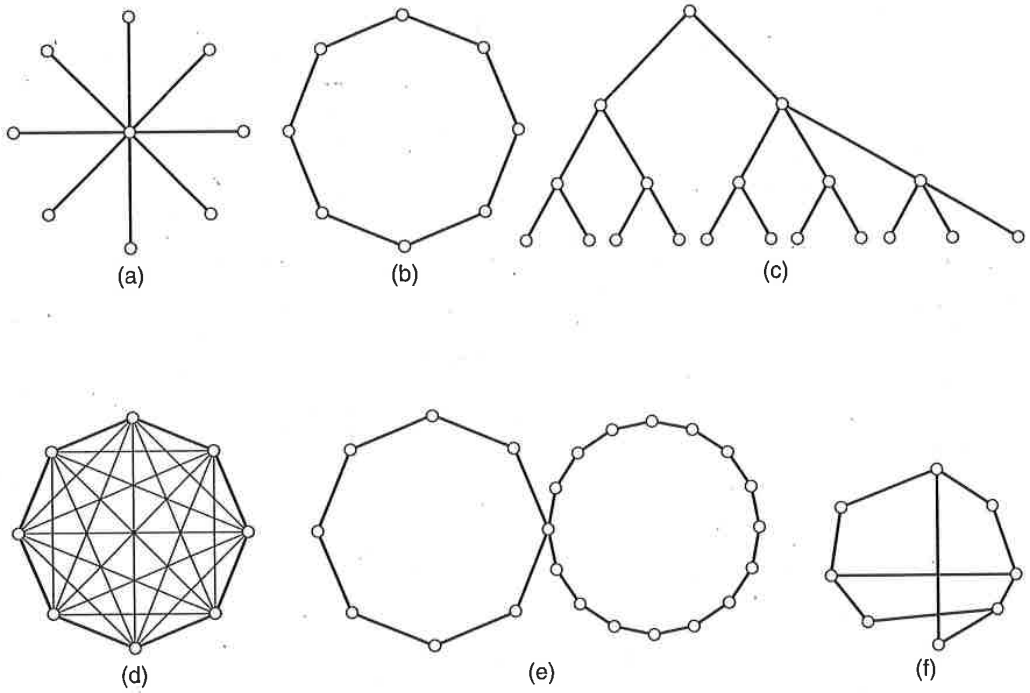


Fig. 1-6. Some possible topologies for a point-to-point subnet. (a) Star. (b) Ring. (c) Tree. (d) Complete. (e) Intersecting rings. (f) Irregular.

A second possibility for a WAN is a satellite or ground radio system. Each router has an antenna through which it can send and receive. All routers can hear the output *from* the satellite, and in some cases they can also hear the upward transmissions of their fellow routers *to* the satellite as well. Sometimes the routers are connected to a substantial point-to-point subnet, with only some of them having a satellite antenna. Satellite networks are inherently broadcast and are most useful when the broadcast property is important.

1.2.4. Wireless Networks

Mobile computers, such as notebook computers and personal digital assistants (PDAs), are the fastest-growing segment of the computer industry. Many of the owners of these computers have desktop machines on LANs and WANs back at the office and want to be connected to their home base even when away from home or en route. Since having a wired connection is impossible in cars and airplanes, there is a lot of interest in wireless networks. In this section we will

briefly introduce this topic. (Note: by section, we mean those portions of the book with a three-part number such as 1.2.4.)

Actually, digital wireless communication is not a new idea. As early as 1901, the Italian physicist Guglielmo Marconi demonstrated a ship-to-shore wireless telegraph using Morse Code (dots and dashes are binary, after all). Modern digital wireless systems have better performance, but the basic idea is the same. Additional information about these systems can be found in (Garg and Wilkes, 1996; and Pahlavan et al., 1995).

Wireless networks have many uses. A common one is the portable office. People on the road often want to use their portable electronic equipment to send and receive telephone calls, faxes, and electronic mail, read remote files, login on remote machines, and so on, and do this from anywhere on land, sea, or air.

Wireless networks are of great value to fleets of trucks, taxis, buses, and repairpersons for keeping in contact with home. Another use is for rescue workers at disaster sites (fires, floods, earthquakes, etc.) where the telephone system has been destroyed. Computers there can send messages, keep records, and so on.

Finally, wireless networks are important to the military. If you have to be able to fight a war anywhere on earth on short notice, counting on using the local networking infrastructure is probably not a good idea. It is better to bring your own.

Although wireless networking and mobile computing are often related, they are not identical, as Fig. 1-7 shows. Portable computers are sometimes wired. For example, if a traveler plugs a portable computer into the telephone jack in a hotel, we have mobility without a wireless network. Another example is someone carrying a portable computer along as he inspects a train for technical problems. Here a long cord can trail along behind (vacuum cleaner model).

Wireless	Mobile	Applications
No	No	Stationary workstations in offices
No	Yes	Using a portable in a hotel; train maintenance
Yes	No	LANs in older, unwired buildings
Yes	Yes	Portable office; PDA for store inventory

Fig. 1-7. Combinations of wireless networks and mobile computing.

On the other hand, some wireless computers are not portable. An important example here is a company that owns an older building that does not have network cabling installed and wants to connect its computers. Installing a wireless LAN may require little more than buying a small box with some electronics and setting up some antennas. This solution may be cheaper than wiring the building.

Although wireless LANs are easy to install, they also have some disadvantages. Typically they have a capacity of 1–2 Mbps, which is much slower than

wired LANs. The error rates are often much higher, too, and the transmissions from different computers can interfere with one another.

But of course, there are also the true mobile, wireless applications, ranging from the portable office to people walking around a store with a PDA doing inventory. At many busy airports, car rental return clerks work out in the parking lot with wireless portable computers. They type in the license plate number of returning cars, and their portable, which has a built-in printer, calls the main computer, gets the rental information, and prints out the bill on the spot. True mobile computing is discussed further in (Forman and Zahorjan, 1994).

Wireless networks come in many forms. Some universities are already installing antennas all over campus to allow students to sit under the trees and consult the library's card catalog. Here the computers communicate directly with the wireless LAN in digital form. Another possibility is using a cellular (i.e., portable) telephone with a traditional analog modem. Direct digital cellular service, called **CDPD (Cellular Digital Packet Data)** is becoming available in many cities. We will study it in Chap. 4.

Finally, it is possible to have different combinations of wired and wireless networking. For example, in Fig. 1-8(a), we depict an airplane with a number of people using modems and seat-back telephones to call the office. Each call is independent of the other ones. A much more efficient option, however, is the flying LAN of Fig. 1-8(b). Here each seat comes equipped with an Ethernet connector into which passengers can plug their computers. A single router on the aircraft maintains a radio link with some router on the ground, changing routers as it flies along. This configuration is just a traditional LAN, except that its connection to the outside world happens to be a radio link instead of a hardwired line.

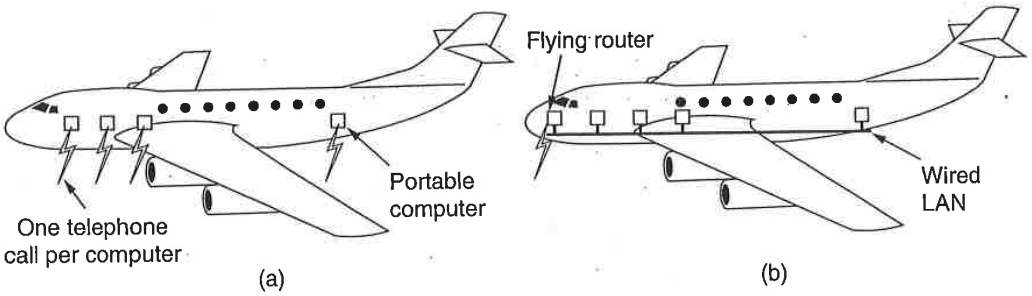


Fig. 1-8. (a) Individual mobile computers. (b) A flying LAN.

While many people believe that wireless portable computers are the wave of the future, at least one dissenting voice has been heard. Bob Metcalfe, the inventor of Ethernet, has written: "Mobile wireless computers are like mobile pipeless bathrooms—portapotties. They will be common on vehicles, and at construction sites, and rock concerts. My advice is to wire up your home and stay there" (Metcalfe, 1995). Will most people follow Metcalfe's advice? Time will tell.

1.2.5. Internetworks

Many networks exist in the world, often with different hardware and software. People connected to one network often want to communicate with people attached to a different one. This desire requires connecting together different, and frequently incompatible networks, sometimes by using machines called **gateways** to make the connection and provide the necessary translation, both in terms of hardware and software. A collection of interconnected networks is called an **internetwork** or just **internet**.

A common form of internet is a collection of LANs connected by a WAN. In fact, if we were to replace the label "subnet" in Fig. 1-5 by "WAN," nothing else in the figure would have to change. The only real distinction between a subnet and a WAN in this case is whether or not hosts are present. If the system within the closed curve contains only routers, it is a subnet. If it contains both routers and hosts with their own users, it is a WAN.

To avoid confusion, please note that the word "internet" will always be used in this book in a generic sense. In contrast, the **Internet** (note uppercase I) means a specific worldwide internet that is widely used to connect universities, government offices, companies, and of late, private individuals. We will have much to say about both internets and the Internet later in this book.

Subnets, networks, and internetworks are often confused. Subnet makes the most sense in the context of a wide area network, where it refers to the collection of routers and communication lines owned by the network operator, for example, companies like America Online and CompuServe. As an analogy, the telephone system consists of telephone switching offices connected to each other by high-speed lines, and to houses and businesses by low-speed lines. These lines and equipment, owned and managed by the telephone company, form the subnet of the telephone system. The telephones themselves (the hosts in this analogy) are not part of the subnet. The combination of a subnet and its hosts forms a network. In the case of a LAN, the cable and the hosts form the network. There really is no subnet.

An internetwork is formed when distinct networks are connected together. In our view, connecting a LAN and a WAN or connecting two LANs forms an internetwork, but there is little agreement in the industry over terminology in this area.

1.3. NETWORK SOFTWARE

The first computer networks were designed with the hardware as the main concern and the software as an afterthought. This strategy no longer works. Network software is now highly structured. In the following sections we examine the software structuring technique in some detail. The method described here forms the keystone of the entire book and will occur repeatedly later on.

1.3.1. Protocol Hierarchies

To reduce their design complexity, most networks are organized as a series of **layers** or **levels**, each one built upon the one below it. The number of layers, the name of each layer, the contents of each layer, and the function of each layer differ from network to network. However, in all networks, the purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented.

Layer n on one machine carries on a conversation with layer n on another machine. The rules and conventions used in this conversation are collectively known as the layer n **protocol**. Basically, a protocol is an agreement between the communicating parties on how communication is to proceed. As an analogy, when a woman is introduced to a man, she may choose to stick out her hand. He, in turn, may decide either to shake it or kiss it, depending, for example, on whether she is an American lawyer at a business meeting or a European princess at a formal ball. Violating the protocol will make communication more difficult, if not impossible.

A five-layer network is illustrated in Fig. 1-9. The entities comprising the corresponding layers on different machines are called **peers**. In other words, it is the peers that communicate using the protocol.

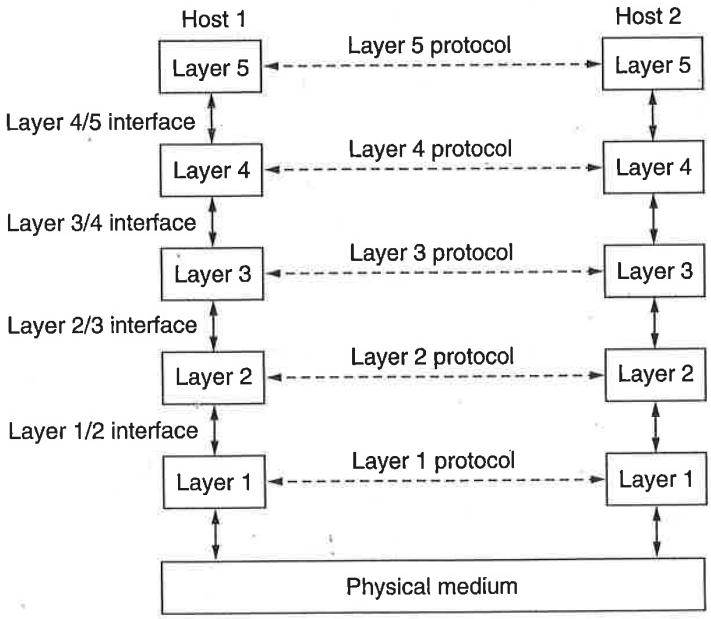


Fig. 1-9. Layers, protocols, and interfaces.

In reality, no data are directly transferred from layer n on one machine to layer n on another machine. Instead, each layer passes data and control

information to the layer immediately below it, until the lowest layer is reached. Below layer 1 is the **physical medium** through which actual communication occurs. In Fig. 1-9, virtual communication is shown by dotted lines and physical communication by solid lines.

Between each pair of adjacent layers there is an **interface**. The interface defines which primitive operations and services the lower layer offers to the upper one. When network designers decide how many layers to include in a network and what each one should do, one of the most important considerations is defining clean interfaces between the layers. Doing so, in turn, requires that each layer perform a specific collection of well-understood functions. In addition to minimizing the amount of information that must be passed between layers, clean-cut interfaces also make it simpler to replace the implementation of one layer with a completely different implementation (e.g., all the telephone lines are replaced by satellite channels), because all that is required of the new implementation is that it offers exactly the same set of services to its upstairs neighbor as the old implementation did.

A set of layers and protocols is called a **network architecture**. The specification of an architecture must contain enough information to allow an implementer to write the program or build the hardware for each layer so that it will correctly obey the appropriate protocol. Neither the details of the implementation nor the specification of the interfaces are part of the architecture because these are hidden away inside the machines and not visible from the outside. It is not even necessary that the interfaces on all machines in a network be the same, provided that each machine can correctly use all the protocols. A list of protocols used by a certain system, one protocol per layer, is called a **protocol stack**. The subjects of network architectures, protocol stacks, and the protocols themselves are the principal topics of this book.

An analogy may help explain the idea of multilayer communication. Imagine two philosophers (peer processes in layer 3), one of whom speaks Urdu and English and one of whom speaks Chinese and French. Since they have no common language, they each engage a translator (peer processes at layer 2), each of whom in turn contacts a secretary (peer processes in layer 1). Philosopher 1 wishes to convey his affection for *oryctolagus cuniculus* to his peer. To do so, he passes a message (in English) across the 2/3 interface, to his translator, saying "I like rabbits," as illustrated in Fig. 1-10. The translators have agreed on a neutral language, Dutch, so the message is converted to "Ik hou van konijnen." The choice of language is the layer 2 protocol and is up to the layer 2 peer processes.

The translator then gives the message to a secretary for transmission, by, for example, fax (the layer 1 protocol). When the message arrives, it is translated into French and passed across the 2/3 interface to philosopher 2. Note that each protocol is completely independent of the other ones as long as the interfaces are not changed. The translators can switch from Dutch to say, Finnish, at will, provided that they both agree, and neither changes his interface with either layer 1 or

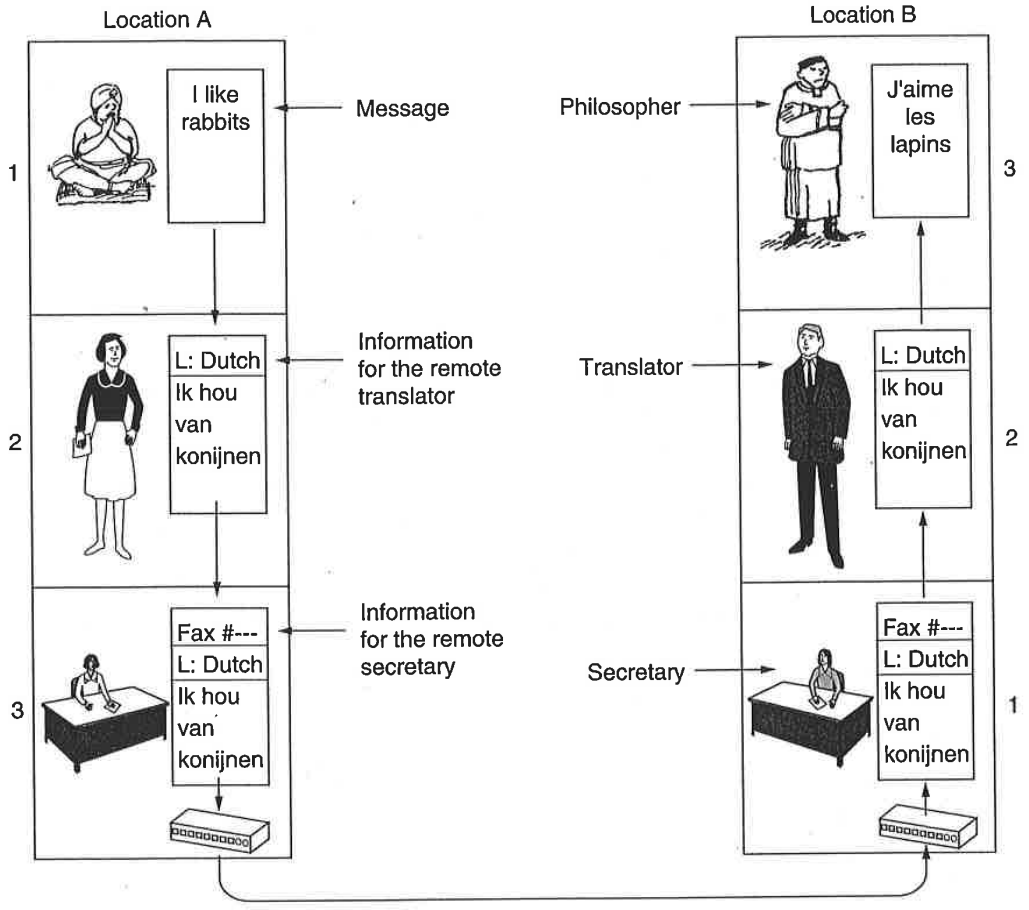


Fig. 1-10. The philosopher-translator-secretary architecture.

layer 3. Similarly the secretaries can switch from fax to email, or telephone without disturbing (or even informing) the other layers. Each process may add some information intended only for its peer. This information is not passed upward to the layer above.

Now consider a more technical example: how to provide communication to the top layer of the five-layer network in Fig. 1-11. A message, *M*, is produced by an application process running in layer 5 and given to layer 4 for transmission. Layer 4 puts a **header** in front of the message to identify the message and passes the result to layer 3. The header includes control information, such as sequence numbers, to allow layer 4 on the destination machine to deliver messages in the right order if the lower layers do not maintain sequence. In some layers, headers also contain sizes, times, and other control fields.

In many networks, there is no limit to the size of messages transmitted in the layer 4 protocol, but there is nearly always a limit imposed by the layer 3 protocol. Consequently, layer 3 must break up the incoming messages into smaller

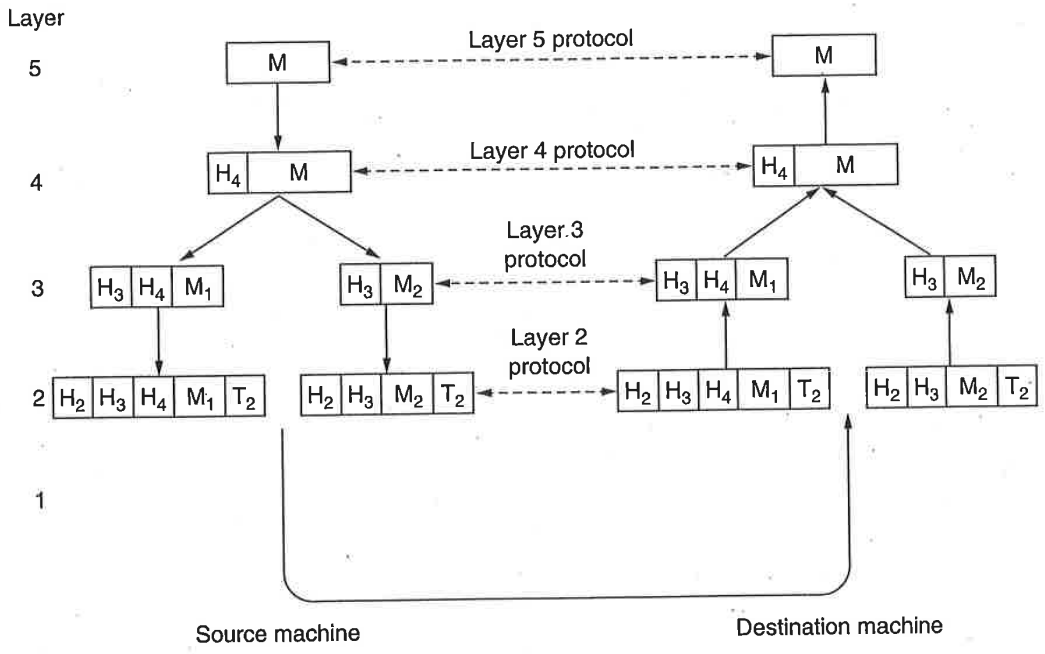


Fig. 1-11. Example information flow supporting virtual communication in layer 5.

units, packets, prepending a layer 3 header to each packet. In this example, M is split into two parts, M_1 and M_2.

Layer 3 decides which of the outgoing lines to use and passes the packets to layer 2. Layer 2 adds not only a header to each piece, but also a trailer, and gives the resulting unit to layer 1 for physical transmission. At the receiving machine the message moves upward, from layer to layer, with headers being stripped off as it progresses. None of the headers for layers below n are passed up to layer n.

The important thing to understand about Fig. 1-11 is the relation between the virtual and actual communication and the difference between protocols and interfaces. The peer processes in layer 4, for example, conceptually think of their communication as being "horizontal," using the layer 4 protocol. Each one is likely to have a procedure called something like *SendToOtherSide* and *GetFromOtherSide*, even though these procedures actually communicate with lower layers across the 3/4 interface, not with the other side.

The peer process abstraction is crucial to all network design. Using it, the unmanageable task of designing the complete network can be broken into several smaller, manageable, design problems, namely the design of the individual layers.

Although Section 1-3 is called "Network Software," it is worth pointing out that the lower layers of a protocol hierarchy are frequently implemented in hardware or firmware. Nevertheless, complex protocol algorithms are involved, even if they are embedded (in whole or in part) in hardware.

1.3.2. Design Issues for the Layers

Some of the key design issues that occur in computer networking are present in several layers. Below, we will briefly mention some of the more important ones.

Every layer needs a mechanism for identifying senders and receivers. Since a network normally has many computers, some of which have multiple processes, a means is needed for a process on one machine to specify with whom it wants to talk. As a consequence of having multiple destinations, some form of addressing is needed in order to specify a specific destination.

Another set of design decisions concerns the rules for data transfer. In some systems, data only travel in one direction (**simplex communication**). In others they can travel in either direction, but not simultaneously (**half-duplex communication**). In still others they travel in both directions at once (**full-duplex communication**). The protocol must also determine how many logical channels the connection corresponds to, and what their priorities are. Many networks provide at least two logical channels per connection, one for normal data and one for urgent data.

Error control is an important issue because physical communication circuits are not perfect. Many error-detecting and error-correcting codes are known, but both ends of the connection must agree on which one is being used. In addition, the receiver must have some way of telling the sender which messages have been correctly received and which have not.

Not all communication channels preserve the order of messages sent on them. To deal with a possible loss of sequencing, the protocol must make explicit provision for the receiver to allow the pieces to be put back together properly. An obvious solution is to number the pieces, but this solution still leaves open the question of what should be done with pieces that arrive out of order.

An issue that occurs at every level is how to keep a fast sender from swamping a slow receiver with data. Various solutions have been proposed and will be discussed later. Some of them involve some kind of feedback from the receiver to the sender, either directly or indirectly, about the receiver's current situation. Others limit the sender to an agreed upon transmission rate.

Another problem that must be solved at several levels is the inability of all processes to accept arbitrarily long messages. This property leads to mechanisms for disassembling, transmitting, and then reassembling messages. A related issue is what to do when processes insist upon transmitting data in units that are so small that sending each one separately is inefficient. Here the solution is to gather together several small messages heading toward a common destination into a single large message and dismember the large message at the other side.

When it is inconvenient or expensive to set up a separate connection for each pair of communicating processes, the underlying layer may decide to use the same connection for multiple, unrelated conversations. As long as this multiplexing and

demultiplexing is done transparently, it can be used by any layer. Multiplexing is needed in the physical layer, for example, where all the traffic for all connections has to be sent over at most a few physical circuits.

When there are multiple paths between source and destination, a route must be chosen. Sometimes this decision must be split over two or more layers. For example, to send data from London to Rome, a high-level decision might have to be made to go via France or Germany based on their respective privacy laws, and a low-level decision might have to be made to choose one of the many available circuits based on the current traffic load.

1.3.3. Interfaces and Services

The function of each layer is to provide services to the layer above it. In this section we will look at precisely what a service is in more detail, but first we will give some terminology.

The active elements in each layer are often called **entities**. An entity can be a software entity (such as a process), or a hardware entity (such as an intelligent I/O chip). Entities in the same layer on different machines are called **peer entities**. The entities in layer n implement a service used by layer $n + 1$. In this case layer n is called the **service provider** and layer $n + 1$ is called the **service user**. Layer n may use the services of layer $n - 1$ in order to provide its service. It may offer several classes of service, for example, fast, expensive communication and slow, cheap communication.

Services are available at **SAP**s (**Service Access Points**), The layer n SAPs are the places where layer $n + 1$ can access the services offered. Each SAP has an address that uniquely identifies it. To make this point clearer, the SAPs in the telephone system are the sockets into which modular telephones can be plugged, and the SAP addresses are the telephone numbers of these sockets. To call someone, you must know the callee's SAP address. Similarly, in the postal system, the SAP addresses are street addresses and post office box numbers. To send a letter, you must know the addressee's SAP address.

In order for two layers to exchange information, there has to be an agreed upon set of rules about the interface. At a typical interface, the layer $n + 1$ entity passes an **IDU** (**Interface Data Unit**) to the layer n entity through the SAP as shown in Fig. 1-12. The IDU consists of an **SDU** (**Service Data Unit**) and some control information. The SDU is the information passed across the network to the peer entity and then up to layer $n + 1$. The control information is needed to help the lower layer do its job (e.g., the number of bytes in the SDU) but is not part of the data itself.

In order to transfer the SDU, the layer n entity may have to fragment it into several pieces, each of which is given a header and sent as a separate **PDU** (**Protocol Data Unit**) such as a packet. The PDU headers are used by the peer entities

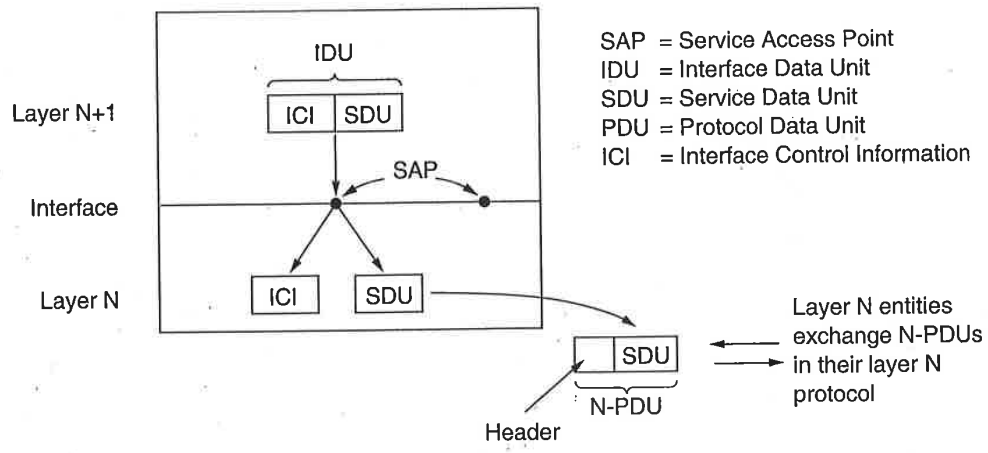


Fig. 1-12. Relation between layers at an interface.

to carry out their peer protocol. They identify which PDUs contain data and which contain control information, provide sequence numbers and counts, and so on.

1.3.4. Connection-Oriented and Connectionless Services

Layers can offer two different types of service to the layers above them: connection-oriented and connectionless. In this section we will look at these two types and examine the differences between them.

Connection-oriented service is modeled after the telephone system. To talk to someone, you pick up the phone, dial the number, talk, and then hang up. Similarly, to use a connection-oriented network service, the service user first establishes a connection, uses the connection, and then releases the connection. The essential aspect of a connection is that it acts like a tube: the sender pushes objects (bits) in at one end, and the receiver takes them out in the same order at the other end.

In contrast, **connectionless service** is modeled after the postal system. Each message (letter) carries the full destination address, and each one is routed through the system independent of all the others. Normally, when two messages are sent to the same destination, the first one sent will be the first one to arrive. However, it is possible that the first one sent can be delayed so that the second one arrives first. With a connection-oriented service this is impossible.

Each service can be characterized by a **quality of service**. Some services are reliable in the sense that they never lose data. Usually, a reliable service is implemented by having the receiver acknowledge the receipt of each message, so the sender is sure that it arrived. The acknowledgement process introduces overhead and delays, which are often worth it but are sometimes undesirable.

A typical situation in which a reliable connection-oriented service is

appropriate is file transfer. The owner of the file wants to be sure that all the bits arrive correctly and in the same order they were sent. Very few file transfer customers would prefer a service that occasionally scrambles or loses a few bits, even if it is much faster.

Reliable connection-oriented service has two minor variations: message sequences and byte streams. In the former, the message boundaries are preserved. When two 1-KB messages are sent, they arrive as two distinct 1-KB messages, never as one 2-KB message. (Note: KB means kilobytes; kb means kilobits.) In the latter, the connection is simply a stream of bytes, with no message boundaries. When 2K bytes arrive at the receiver, there is no way to tell if they were sent as one 2-KB message, two 1-KB messages, or 2048 1-byte messages. If the pages of a book are sent over a network to a phototypesetter as separate messages, it might be important to preserve the message boundaries. On the other hand, with a terminal logging into a remote timesharing system, a byte stream from the terminal to the computer is all that is needed.

As mentioned above, for some applications, the delays introduced by acknowledgements are unacceptable. One such application is digitized voice traffic. It is preferable for telephone users to hear a bit of noise on the line or a garbled word from time to time than to introduce a delay to wait for acknowledgements. Similarly, when transmitting a video film, having a few pixels wrong is no problem, but having the film jerk along as the flow stops to correct errors is very irritating.

Not all applications require connections. For example, as electronic mail becomes more common, can electronic junk mail be far behind? The electronic junk mail sender probably does not want to go to the trouble of setting up and later tearing down a connection just to send one item. Nor is 100 percent reliable delivery essential, especially if it costs more. All that is needed is a way to send a single message that has a high probability of arrival, but no guarantee. Unreliable (meaning not acknowledged) connectionless service is often called **datagram service**, in analogy with telegram service, which also does not provide an acknowledgement back to the sender.

In other situations, the convenience of not having to establish a connection to send one short message is desired, but reliability is essential. The **acknowledged datagram service** can be provided for these applications. It is like sending a registered letter and requesting a return receipt. When the receipt comes back, the sender is absolutely sure that the letter was delivered to the intended party and not lost along the way.

Still another service is the **request-reply service**. In this service the sender transmits a single datagram containing a request; the reply contains the answer. For example, a query to the local library asking where Uighur is spoken falls into this category. Request-reply is commonly used to implement communication in the client-server model: the client issues a request and the server responds to it. Figure 1-13 summarizes the types of services discussed above.

	Service	Example
Connection-oriented	Reliable message stream	Sequence of pages
	Reliable byte stream	Remote login
	Unreliable connection	Digitized voice
Connection-less	Unreliable datagram	Electronic junk mail
	Acknowledged datagram	Registered mail
	Request-reply	Database query

Fig. 1-13. Six different types of service.

1.3.5. Service Primitives

A service is formally specified by a set of **primitives** (operations) available to a user or other entity to access the service. These primitives tell the service to perform some action or report on an action taken by a peer entity. One way to classify the service primitives is to divide them into four classes as shown in Fig. 1-14.

Primitive	Meaning
Request	An entity wants the service to do some work
Indication	An entity is to be informed about an event
Response	An entity wants to respond to an event
Confirm	The response to an earlier request has come back

Fig. 1-14. Four classes of service primitives.

To illustrate the uses of the primitives, consider how a connection is established and released. The initiating entity does a CONNECT.request which results in a packet being sent. The receiver then gets a CONNECT.indication announcing that an entity somewhere wants to set up a connection to it. The entity getting the CONNECT.indication then uses the CONNECT.response primitive to tell whether it wants to accept or reject the proposed connection. Either way, the entity issuing the initial CONNECT.request finds out what happened via a CONNECT.confirm primitive.

Primitives can have parameters, and most of them do. The parameters to a CONNECT.request might specify the machine to connect to, the type of service desired, and the maximum message size to be used on the connection. The parameters to a CONNECT.indication might contain the caller's identity, the type of

service desired, and the proposed maximum message size. If the called entity did not agree to the proposed maximum message size, it could make a counterproposal in its *response* primitive, which would be made available to the original caller in the *confirm*. The details of this **negotiation** are part of the protocol. For example, in the case of two conflicting proposals about maximum message size, the protocol might specify that the smaller value is always chosen.

As an aside on terminology, we will carefully avoid the terms "open a connection" and "close a connection" because to electrical engineers, an "open circuit" is one with a gap or break in it. Electricity can only flow over "closed circuits." Computer scientists would never agree to having information flow over a closed circuit. To keep both camps pacified, we will use the terms "establish a connection" and "release a connection."

Services can be either **confirmed** or **unconfirmed**. In a confirmed service, there is a *request*, an *indication*, a *response*, and a *confirm*. In an unconfirmed service, there is just a *request* and an *indication*. CONNECT is always a confirmed service because the remote peer must agree to establish a connection. Data transfer, on the other hand, can be either confirmed or unconfirmed, depending on whether or not the sender needs an acknowledgement. Both kinds of services are used in networks.

To make the concept of a service more concrete, let us consider as an example a simple connection-oriented service with eight service primitives as follows:

1. CONNECT.request – Request a connection to be established.
2. CONNECT.indication – Signal the called party.
3. CONNECT.response – Used by the callee to accept/reject calls.
4. CONNECT.confirm – Tell the caller whether the call was accepted.
5. DATA.request – Request that data be sent.
6. DATA.indication – Signal the arrival of data.
7. DISCONNECT.request – Request that a connection be released.
8. DISCONNECT.indication – Signal the peer about the request.

In this example, CONNECT is a confirmed service (an explicit response is required), whereas DISCONNECT is unconfirmed (no response).

It may be helpful to make an analogy with the telephone system to see how these primitives are used. For this analogy, consider the steps required to call Aunt Millie on the telephone and invite her to your house for tea.

1. CONNECT.request – Dial Aunt Millie's phone number.
2. CONNECT.indication – Her phone rings.
3. CONNECT.response – She picks up the phone.

4. CONNECT.confirm – You hear the ringing stop.
5. DATA.request – You invite her to tea.
6. DATA.indication – She hears your invitation.
7. DATA.request – She says she would be delighted to come.
8. DATA.indication – You hear her acceptance.
9. DISCONNECT.request – You hang up the phone.
10. DISCONNECT.indication – She hears it and hangs up too.

Figure 1-15 shows this same sequence of steps as a series of service primitives, including the final confirmation of disconnection. Each step involves an interaction between two layers on one of the computers. Each *request* or *response* causes an *indication* or *confirm* at the other side a little later. In this example, the service users (you and Aunt Millie) are in layer $N + 1$ and the service provider (the telephone system) is in layer N.

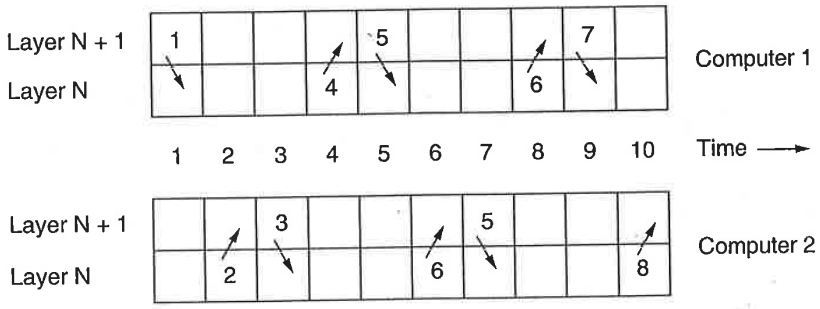


Fig. 1-15. How a computer would invite its Aunt Millie to tea. The numbers near the tail end of each arrow refer to the eight service primitives discussed in this section.

1.3.6. The Relationship of Services to Protocols

Services and protocols are distinct concepts, although they are frequently confused. This distinction is so important, however, that we emphasize it again here. A *service* is a set of primitives (operations) that a layer provides to the layer above it. The service defines what operations the layer is prepared to perform on behalf of its users, but it says nothing at all about how these operations are implemented. A service relates to an interface between two layers, with the lower layer being the service provider and the upper layer being the service user.

A *protocol*, in contrast, is a set of rules governing the format and meaning of the frames, packets, or messages that are exchanged by the peer entities within a layer. Entities use protocols in order to implement their service definitions. They

are free to change their protocols at will, provided they do not change the service visible to their users. In this way, the service and the protocol are completely decoupled.

An analogy with programming languages is worth making. A service is like an abstract data type or an object in an object-oriented language. It defines operations that can be performed on an object but does not specify how these operations are implemented. A protocol relates to the *implementation* of the service and as such is not visible to the user of the service.

Many older protocols did not distinguish the service from the protocol. In effect, a typical layer might have had a service primitive SEND PACKET with the user providing a pointer to a fully assembled packet. This arrangement meant that all changes to the protocol were immediately visible to the users. Most network designers now regard such a design as a serious blunder.

1.4. REFERENCE MODELS

Now that we have discussed layered networks in the abstract, it is time to look at some examples. In the next two sections we will discuss two important network architectures, the OSI reference model and the TCP/IP reference model.

1.4.1. The OSI Reference Model

The OSI model is shown in Fig. 1-16 (minus the physical medium). This model is based on a proposal developed by the International Standards Organization (ISO) as a first step toward international standardization of the protocols used in the various layers (Day and Zimmermann, 1983). The model is called the **ISO OSI (Open Systems Interconnection) Reference Model** because it deals with connecting open systems—that is, systems that are open for communication with other systems. We will usually just call it the OSI model for short.

The OSI model has seven layers. The principles that were applied to arrive at the seven layers are as follows:

1. A layer should be created where a different level of abstraction is needed.
2. Each layer should perform a well defined function.
3. The function of each layer should be chosen with an eye toward defining internationally standardized protocols.
4. The layer boundaries should be chosen to minimize the information flow across the interfaces.
5. The number of layers should be large enough that distinct functions need not be thrown together in the same layer out of necessity, and small enough that the architecture does not become unwieldy.

Below we will discuss each layer of the model in turn, starting at the bottom layer. Note that the OSI model itself is not a network architecture because it does not specify the exact services and protocols to be used in each layer. It just tells what each layer should do. However, ISO has also produced standards for all the layers, although these are not part of the reference model itself. Each one has been published as a separate international standard.

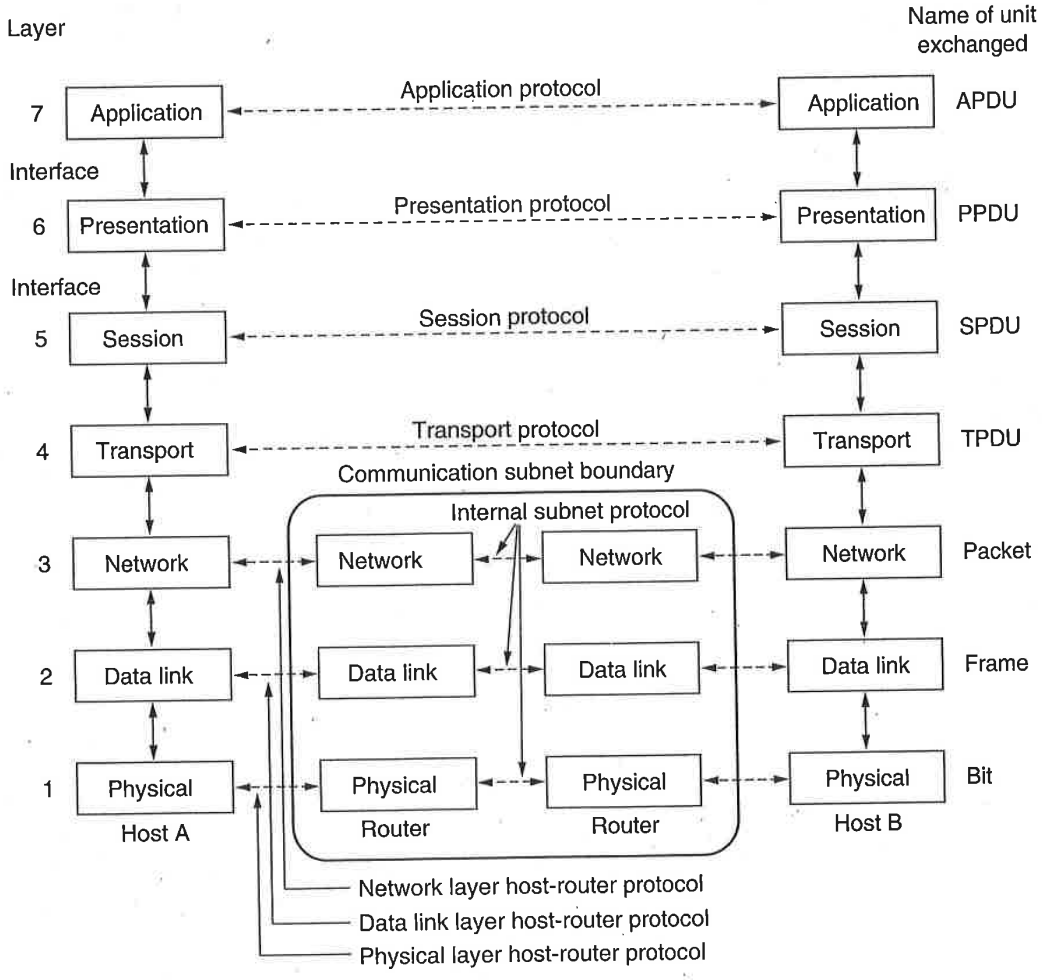


Fig. 1-16. The OSI reference model.

The Physical Layer

The **physical layer** is concerned with transmitting raw bits over a communication channel. The design issues have to do with making sure that when one side sends a 1 bit, it is received by the other side as a 1 bit, not as a 0 bit. Typical

questions here are how many volts should be used to represent a 1 and how many for a 0, how many microseconds a bit lasts, whether transmission may proceed simultaneously in both directions, how the initial connection is established and how it is torn down when both sides are finished, and how many pins the network connector has and what each pin is used for. The design issues here largely deal with mechanical, electrical, and procedural interfaces, and the physical transmission medium, which lies below the physical layer.

The Data Link Layer

The main task of the **data link layer** is to take a raw transmission facility and transform it into a line that appears free of undetected transmission errors to the network layer. It accomplishes this task by having the sender break the input data up into **data frames** (typically a few hundred or a few thousand bytes), transmit the frames sequentially, and process the **acknowledgement frames** sent back by the receiver. Since the physical layer merely accepts and transmits a stream of bits without any regard to meaning or structure, it is up to the data link layer to create and recognize frame boundaries. This can be accomplished by attaching special bit patterns to the beginning and end of the frame. If these bit patterns can accidentally occur in the data, special care must be taken to make sure these patterns are not incorrectly interpreted as frame delimiters.

A noise burst on the line can destroy a frame completely. In this case, the data link layer software on the source machine can retransmit the frame. However, multiple transmissions of the same frame introduce the possibility of duplicate frames. A duplicate frame could be sent if the acknowledgement frame from the receiver back to the sender were lost. It is up to this layer to solve the problems caused by damaged, lost, and duplicate frames. The data link layer may offer several different service classes to the network layer, each of a different quality and with a different price.

Another issue that arises in the data link layer (and most of the higher layers as well) is how to keep a fast transmitter from drowning a slow receiver in data. Some traffic regulation mechanism must be employed to let the transmitter know how much buffer space the receiver has at the moment. Frequently, this flow regulation and the error handling are integrated.

If the line can be used to transmit data in both directions, this introduces a new complication that the data link layer software must deal with. The problem is that the acknowledgement frames for A to B traffic compete for the use of the line with data frames for the B to A traffic. A clever solution (piggybacking) has been devised; we will discuss it in detail later.

Broadcast networks have an additional issue in the data link layer: how to control access to the shared channel. A special sublayer of the data link layer, the medium access sublayer, deals with this problem.

The Network Layer

The **network layer** is concerned with controlling the operation of the subnet. A key design issue is determining how packets are routed from source to destination. Routes can be based on static tables that are "wired into" the network and rarely changed. They can also be determined at the start of each conversation, for example a terminal session. Finally, they can be highly dynamic, being determined anew for each packet, to reflect the current network load.

If too many packets are present in the subnet at the same time, they will get in each other's way, forming bottlenecks. The control of such congestion also belongs to the network layer.

Since the operators of the subnet may well expect remuneration for their efforts, there is often some accounting function built into the network layer. At the very least, the software must count how many packets or characters or bits are sent by each customer, to produce billing information. When a packet crosses a national border, with different rates on each side, the accounting can become complicated.

When a packet has to travel from one network to another to get to its destination, many problems can arise. The addressing used by the second network may be different from the first one. The second one may not accept the packet at all because it is too large. The protocols may differ, and so on. It is up to the network layer to overcome all these problems to allow heterogeneous networks to be interconnected.

In broadcast networks, the routing problem is simple, so the network layer is often thin or even nonexistent.

The Transport Layer

The basic function of the **transport layer** is to accept data from the session layer, split it up into smaller units if need be, pass these to the network layer, and ensure that the pieces all arrive correctly at the other end. Furthermore, all this must be done efficiently, and in a way that isolates the upper layers from the inevitable changes in the hardware technology.

Under normal conditions, the transport layer creates a distinct network connection for each transport connection required by the session layer. If the transport connection requires a high throughput, however, the transport layer might create multiple network connections, dividing the data among the network connections to improve throughput. On the other hand, if creating or maintaining a network connection is expensive, the transport layer might multiplex several transport connections onto the same network connection to reduce the cost. In all cases, the transport layer is required to make the multiplexing transparent to the session layer.

The transport layer also determines what type of service to provide the session

layer, and ultimately, the users of the network. The most popular type of transport connection is an error-free point-to-point channel that delivers messages or bytes in the order in which they were sent. However, other possible kinds of transport service are transport of isolated messages with no guarantee about the order of delivery, and broadcasting of messages to multiple destinations. The type of service is determined when the connection is established.

The transport layer is a true end-to-end layer, from source to destination. In other words, a program on the source machine carries on a conversation with a similar program on the destination machine, using the message headers and control messages. In the lower layers, the protocols are between each machine and its immediate neighbors, and not by the ultimate source and destination machines, which may be separated by many routers. The difference between layers 1 through 3, which are chained, and layers 4 through 7, which are end-to-end, is illustrated in Fig. 1-16.

Many hosts are multiprogrammed, which implies that multiple connections will be entering and leaving each host. There needs to be some way to tell which message belongs to which connection. The transport header (H_4 in Fig. 1-11) is one place this information can be put.

In addition to multiplexing several message streams onto one channel, the transport layer must take care of establishing and deleting connections across the network. This requires some kind of naming mechanism, so that a process on one machine has a way of describing with whom it wishes to converse. There must also be a mechanism to regulate the flow of information, so that a fast host cannot overrun a slow one. Such a mechanism is called **flow control** and plays a key role in the transport layer (also in other layers). Flow control between hosts is distinct from flow control between routers, although we will later see that similar principles apply to both.

The Session Layer

The session layer allows users on different machines to establish **sessions** between them. A session allows ordinary data transport, as does the transport layer, but it also provides enhanced services useful in some applications. A session might be used to allow a user to log into a remote timesharing system or to transfer a file between two machines.

One of the services of the session layer is to manage dialogue control. Sessions can allow traffic to go in both directions at the same time, or in only one direction at a time. If traffic can only go one way at a time (analogous to a single railroad track), the session layer can help keep track of whose turn it is.

A related session service is **token management**. For some protocols, it is essential that both sides do not attempt the same operation at the same time. To manage these activities, the session layer provides tokens that can be exchanged. Only the side holding the token may perform the critical operation.

Another session service is **synchronization**. Consider the problems that might occur when trying to do a 2-hour file transfer between two machines with a 1-hour mean time between crashes. After each transfer was aborted, the whole transfer would have to start over again and would probably fail again the next time as well. To eliminate this problem, the session layer provides a way to insert checkpoints into the data stream, so that after a crash, only the data transferred after the last checkpoint have to be repeated.

The Presentation Layer

The **presentation layer** performs certain functions that are requested sufficiently often to warrant finding a general solution for them, rather than letting each user solve the problems. In particular, unlike all the lower layers, which are just interested in moving bits reliably from here to there, the presentation layer is concerned with the syntax and semantics of the information transmitted.

A typical example of a presentation service is encoding data in a standard agreed upon way. Most user programs do not exchange random binary bit strings. They exchange things such as people's names, dates, amounts of money, and invoices. These items are represented as character strings, integers, floating-point numbers, and data structures composed of several simpler items. Different computers have different codes for representing character strings (e.g., ASCII and Unicode), integers (e.g., one's complement and two's complement), and so on. In order to make it possible for computers with different representations to communicate, the data structures to be exchanged can be defined in an abstract way, along with a standard encoding to be used "on the wire." The presentation layer manages these abstract data structures and converts from the representation used inside the computer to the network standard representation and back.

The Application Layer

The **application layer** contains a variety of protocols that are commonly needed. For example, there are hundreds of incompatible terminal types in the world. Consider the plight of a full screen editor that is supposed to work over a network with many different terminal types, each with different screen layouts, escape sequences for inserting and deleting text, moving the cursor, etc.

One way to solve this problem is to define an abstract **network virtual terminal** that editors and other programs can be written to deal with. To handle each terminal type, a piece of software must be written to map the functions of the network virtual terminal onto the real terminal. For example, when the editor moves the virtual terminal's cursor to the upper left-hand corner of the screen, this software must issue the proper command sequence to the real terminal to get its cursor there too. All the virtual terminal software is in the application layer.

Another application layer function is file transfer. Different file systems have

different file naming conventions, different ways of representing text lines, and so on. Transferring a file between two different systems requires handling these and other incompatibilities. This work, too, belongs to the application layer, as do electronic mail, remote job entry, directory lookup, and various other general-purpose and special-purpose facilities.

Data Transmission in the OSI Model

Figure 1-17 shows an example of how data can be transmitted using the OSI model. The sending process has some data it wants to send to the receiving process. It gives the data to the application layer, which then attaches the application header, *AH* (which may be null), to the front of it and gives the resulting item to the presentation layer.

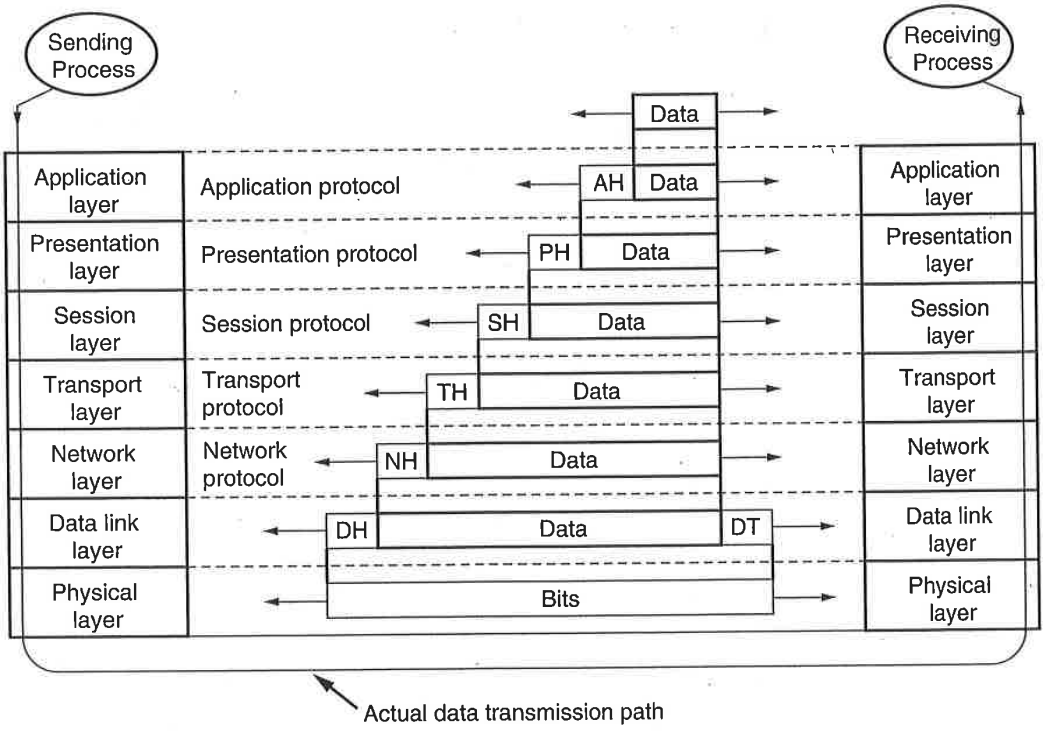


Fig. 1-17. An example of how the OSI model is used. Some of the headers may be null. (Source: H.C. Folts. Used with permission.)

The presentation layer may transform this item in various ways and possibly add a header to the front, giving the result to the session layer. It is important to realize that the presentation layer is not aware of which portion of the data given to it by the application layer is *AH*, if any, and which is true user data.

This process is repeated until the data reach the physical layer, where they are actually transmitted to the receiving machine. On that machine the various

headers are stripped off one by one as the message propagates up the layers until it finally arrives at the receiving process.

The key idea throughout is that although actual data transmission is vertical in Fig. 1-17, each layer is programmed as though it were horizontal. When the sending transport layer, for example, gets a message from the session layer, it attaches a transport header and sends it to the receiving transport layer. From its point of view, the fact that it must actually hand the message to the network layer on its own machine is an unimportant technicality. As an analogy, when a Tagalog-speaking diplomat is addressing the United Nations, he thinks of himself as addressing the other assembled diplomats. That, in fact, he is really only speaking to his translator is seen as a technical detail.

1.4.2. The TCP/IP Reference Model

Let us now turn from the OSI reference model to the reference model used in the grandparent of all computer networks, the ARPANET, and its successor, the worldwide Internet. Although we will give a brief history of the ARPANET later, it is useful to mention a few key aspects of it now. The ARPANET was a research network sponsored by the DoD (U.S. Department of Defense). It eventually connected hundreds of universities and government installations using leased telephone lines. When satellite and radio networks were added later, the existing protocols had trouble interworking with them, so a new reference architecture was needed. Thus the ability to connect multiple networks together in a seamless way was one of the major design goals from the very beginning. This architecture later became known as the **TCP/IP Reference Model**, after its two primary protocols. It was first defined in (Cerf and Kahn, 1974). A later perspective is given in (Leiner et al., 1985). The design philosophy behind the model is discussed in (Clark, 1988).

Given the DoD's worry that some of its precious hosts, routers, and internetwork gateways might get blown to pieces at a moment's notice, another major goal was that the network be able to survive loss of subnet hardware, with existing conversations not being broken off. In other words, DoD wanted connections to remain intact as long as the source and destination machines were functioning, even if some of the machines or transmission lines in between were suddenly put out of operation. Furthermore, a flexible architecture was needed, since applications with divergent requirements were envisioned, ranging from transferring files to real-time speech transmission.

The Internet Layer

All these requirements led to the choice of a packet-switching network based on a connectionless internetwork layer. This layer, called the **internet layer**, is the linchpin that holds the whole architecture together. Its job is to permit hosts to

inject packets into any network and have them travel independently to the destination (potentially on a different network). They may even arrive in a different order than they were sent, in which case it is the job of higher layers to rearrange them, if in-order delivery is desired. Note that "internet" is used here in a generic sense, even though this layer is present in the Internet.

The analogy here is with the (snail) mail system. A person can drop a sequence of international letters into a mail box in one country, and with a little luck, most of them will be delivered to the correct address in the destination country. Probably the letters will travel through one or more international mail gateways along the way, but this is transparent to the users. Furthermore, that each country (i.e., each network) has its own stamps, preferred envelope sizes, and delivery rules is hidden from the users.

The internet layer defines an official packet format and protocol called **IP** (**Internet Protocol**). The job of the internet layer is to deliver IP packets where they are supposed to go. Packet routing is clearly the major issue here, as is avoiding congestion. For these reasons, it is reasonable to say that the TCP/IP internet layer is very similar in functionality to the OSI network layer. Figure 1-18 shows this correspondence.

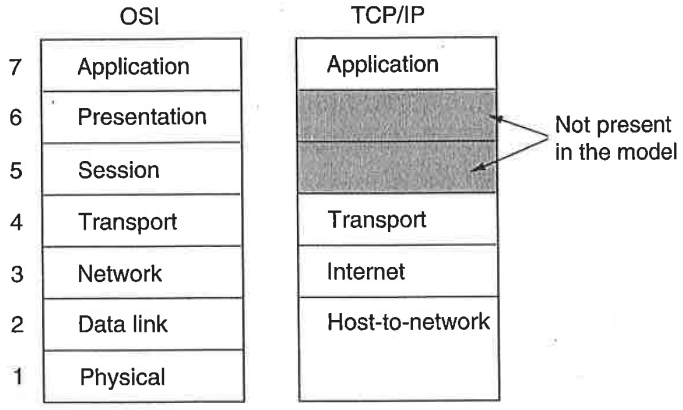


Fig. 1-18. The TCP/IP reference model.

The Transport Layer

The layer above the internet layer in the TCP/IP model is now usually called the **transport layer**. It is designed to allow peer entities on the source and destination hosts to carry on a conversation, the same as in the OSI transport layer. Two end-to-end protocols have been defined here. The first one, **TCP** (**Transmission Control Protocol**) is a reliable connection-oriented protocol that allows a byte stream originating on one machine to be delivered without error on

any other machine in the internet. It fragments the incoming byte stream into discrete messages and passes each one onto the internet layer. At the destination, the receiving TCP process reassembles the received messages into the output stream. TCP also handles flow control to make sure a fast sender cannot swamp a slow receiver with more messages than it can handle.

The second protocol in this layer, **UDP (User Datagram Protocol)**, is an unreliable, connectionless protocol for applications that do not want TCP's sequencing or flow control and wish to provide their own. It is also widely used for one-shot, client-server type request-reply queries and applications in which prompt delivery is more important than accurate delivery, such as transmitting speech or video. The relation of IP, TCP, and UDP is shown in Fig. 1-19. Since the model was developed, IP has been implemented on many other networks.

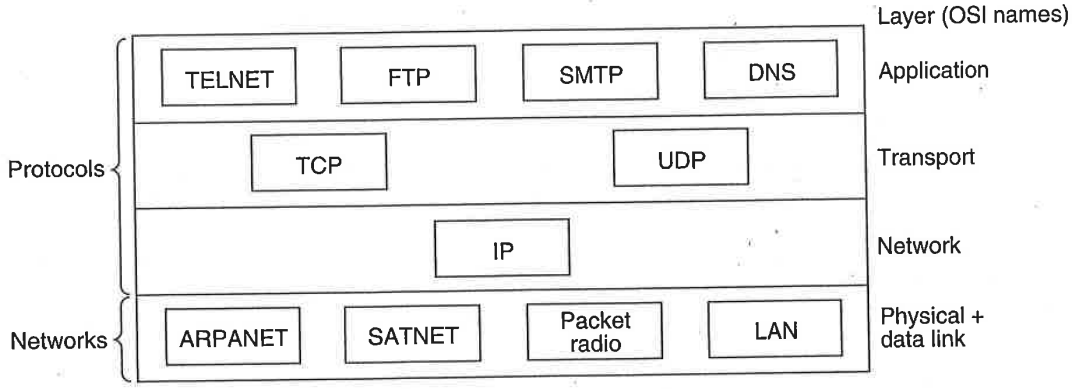


Fig. 1-19. Protocols and networks in the TCP/IP model initially.

The Application Layer

The TCP/IP model does not have session or presentation layers. No need for them was perceived, so they were not included. Experience with the OSI model has proven this view correct: they are of little use to most applications.

On top of the transport layer is the **application layer**. It contains all the higher-level protocols. The early ones included virtual terminal (TELNET), file transfer (FTP), and electronic mail (SMTP), as shown in Fig. 1-19. The virtual terminal protocol allows a user on one machine to log into a distant machine and work there. The file transfer protocol provides a way to move data efficiently from one machine to another. Electronic mail was originally just a kind of file transfer, but later a specialized protocol was developed for it. Many other protocols have been added to these over the years, such as the Domain Name Service (DNS) for mapping host names onto their network addresses, NNTP, the protocol used for moving news articles around, and HTTP, the protocol used for fetching pages on the World Wide Wide, and many others.

The Host-to-Network Layer

Below the internet layer is a great void. The TCP/IP reference model does not really say much about what happens here, except to point out that the host has to connect to the network using some protocol so it can send IP packets over it. This protocol is not defined and varies from host to host and network to network. Books and papers about the TCP/IP model rarely discuss it.

1.4.3. A Comparison of the OSI and TCP Reference Models

The OSI and TCP/IP reference models have much in common. Both are based on the concept of a stack of independent protocols. Also, the functionality of the layers is roughly similar. For example, in both models the layers up through and including the transport layer are there to provide an end-to-end network-independent transport service to processes wishing to communicate. These layers form the transport provider. Again in both models, the layers above transport are application-oriented users of the transport service.

Despite these fundamental similarities, the two models also have many differences. In this section we will focus on the key differences between the two reference models. It is important to note that we are comparing the *reference models* here, not the corresponding *protocol stacks*. The protocols themselves will be discussed later. For an entire book comparing and contrasting TCP/IP and OSI, see (Piscitello and Chapin, 1993).

Three concepts are central to the OSI model:

1. Services
2. Interfaces
3. Protocols

Probably the biggest contribution of the OSI model is to make the distinction between these three concepts explicit. Each layer performs some services for the layer above it. The *service* definition tells what the layer does, not how entities above it access it or how the layer works.

A layer's *interface* tells the processes above it how to access it. It specifies what the parameters are and what results to expect. It, too, says nothing about how the layer works inside.

Finally, the peer *protocols* used in a layer are the layer's own business. It can use any protocols it wants to, as long as it gets the job done (i.e., provides the offered services). It can also change them at will without affecting software in higher layers.

These ideas fit very nicely with modern ideas about object-oriented programming. An object, like a layer, has a set of methods (operations) that processes

outside the object can invoke. The semantics of these methods define the set of services that the object offers. The methods' parameters and results form the object's interface. The code internal to the object is its protocol and is not visible or of any concern outside the object.

The TCP/IP model did not originally clearly distinguish between service, interface, and protocol, although people have tried to retrofit it after the fact to make it more OSI-like. For example, the only real services offered by the internet layer are SEND IP PACKET and RECEIVE IP PACKET.

As a consequence, the protocols in the OSI model are better hidden than in the TCP/IP model and can be replaced relatively easily as the technology changes. Being able to make such changes is one of the main purposes of having layered protocols in the first place.

The OSI reference model was devised *before* the protocols were invented. This ordering means that the model was not biased toward one particular set of protocols, which made it quite general. The down side of this ordering is that the designers did not have much experience with the subject and did not have a good idea of which functionality to put in which layer.

For example, the data link layer originally dealt only with point-to-point networks. When broadcast networks came around, a new sublayer had to be hacked into the model. When people started to build real networks using the OSI model and existing protocols, it was discovered that they did not match the required service specifications (wonder of wonders), so convergence sublayers had to be grafted onto the model to provide a place for papering over the differences. Finally, the committee originally expected that each country would have one network, run by the government and using the OSI protocols, so no thought was given to internetworking. To make a long story short, things did not turn out that way.

With the TCP/IP the reverse was true: the protocols came first, and the model was really just a description of the existing protocols. There was no problem with the protocols fitting the model. They fit perfectly. The only trouble was that the *model* did not fit any other protocol stacks. Consequently, it was not especially useful for describing other non-TCP/IP networks.

Turning from philosophical matters to more specific ones, an obvious difference between the two models is the number of layers: the OSI model has seven layers and the TCP/IP has four layers. Both have (inter)network, transport, and application layers, but the other layers are different.

Another difference is in the area of connectionless versus connection-oriented communication. The OSI model supports both connectionless and connection-oriented communication in the network layer, but only connection-oriented communication in the transport layer, where it counts (because the transport service is visible to the users). The TCP/IP model has only one mode in the network layer (connectionless) but supports both modes in the transport layer, giving the users a choice. This choice is especially important for simple request-response protocols.

1.4.4. A Critique of the OSI Model and Protocols

Neither the OSI model and its protocols nor the TCP/IP model and its protocols are perfect. Quite a bit of criticism can be, and has been, directed at both of them. In this section and the next one, we will look at some of these criticisms. We will begin with OSI and examine TCP/IP afterward.

At the time the second edition of this book was published (1989), it appeared to most experts in the field that the OSI model and its protocols were going to take over the world and push everything else out of their way. This did not happen. Why? A look back at some of the lessons may be useful. These lessons can be summarized as:

1. Bad timing.
2. Bad technology.
3. Bad implementations.
4. Bad politics.

Bad Timing

First let us look at reason one: bad timing. The time at which a standard is established is absolutely critical to its success. David Clark of M.I.T. has a theory of standards that he calls the *apocalypse of the two elephants*, and which is illustrated in Fig. 1-20.

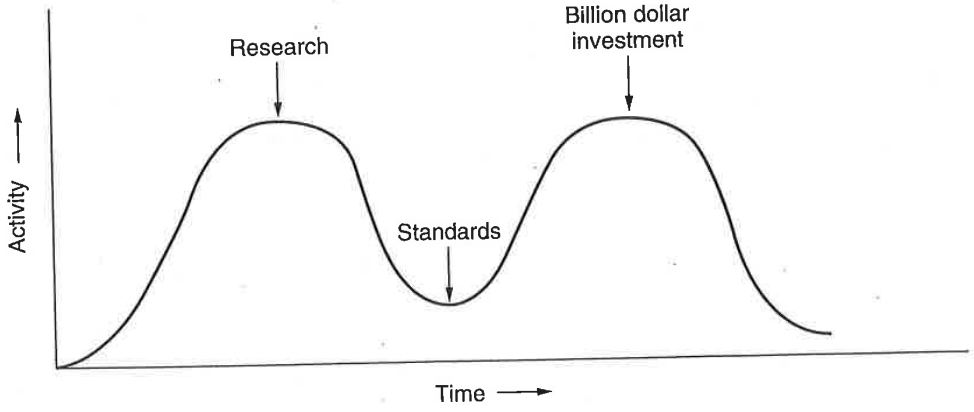


Fig. 1-20. The apocalypse of the two elephants.

This figure shows the amount of activity surrounding a new subject. When the subject is first discovered, there is a burst of research activity in the form of discussions, papers, and meetings. After a while this subsides, corporations discover the subject, and the billion-dollar wave of investment hits.

It is essential that the standards be written in the trough between the two "elephants." If they are written too early, before the research is finished, the subject may still be poorly understood, which leads to bad standards. If they are written too late, so many companies may have already made major investments in different ways of doing things that the standards are effectively ignored. If the interval between the two elephants is very short (because everyone is in a hurry to get started), the people developing the standards may get crushed.

It now appears that the standard OSI protocols got crushed. The competing TCP/IP protocols were already in widespread use by research universities by the time the OSI protocols appeared. While the billion-dollar wave of investment had not yet hit, the academic market was large enough that many vendors had begun cautiously offering TCP/IP products. When OSI came around, they did not want to support a second protocol stack until they were forced to, so there were no initial offerings. With every company waiting for every other company to go first, no company went first and OSI never happened.

Bad Technology

The second reason that OSI never caught on is that both the model and the protocols are flawed. Most discussions of the seven-layer model give the impression that the number and contents of the layers eventually chosen were the only way, or at least the obvious way. This is far from true. The session layer has little use in most applications, and the presentation layer is nearly empty. In fact, the British proposal to ISO only had five layers, not seven. In contrast to the session and presentation layers, the data link and network layers are so full that subsequent work has split them into multiple sublayers, each with different functions.

Although hardly anyone ever admits it in public, the real reason that the OSI model has seven layers is that at the time it was designed, IBM had a proprietary seven-layer protocol called **SNA**™ (**Systems Network Architecture**). At that time, IBM so dominated the computer industry that everyone else, including telephone companies, competing computer companies, and even major governments, were scared to death that IBM would use its market clout to effectively force everybody to use SNA, which it could change whenever it wished. The idea behind OSI was to produce an IBM-like reference model and protocol stack that would become the world standard, and controlled not by one company, but by a neutral organization, ISO.

The OSI model, along with the associated service definitions and protocols, is extraordinarily complex. When piled up, the printed standards occupy a significant fraction of a meter of paper. They are also difficult to implement and inefficient in operation. In this context, a riddle posed by Paul Mockapetris and cited in (Rose, 1993) comes to mind:

Q: What do you get when you cross a mobster with an international standard?
A: Someone who makes you an offer you can't understand.

In addition to being incomprehensible, another problem with OSI is that some functions, such as addressing, flow control, and error control reappear again and again in each layer. Saltzer et al. (1984), for example, have pointed out that to be effective, error control must be done in the highest layer, so that repeating it over and over in each of the lower layers is often unnecessary and inefficient.

Another issue is that the decision to place certain features in particular layers is not always obvious. The virtual terminal handling (now in the application layer) was in the presentation layer during much of the development of the standard. It was moved to the application layer because the committee had trouble deciding what the presentation layer was good for. Data security and encryption were so controversial that no one could agree which layer to put them in, so they were left out altogether. Network management was also omitted from the model for similar reasons.

Another criticism of the original standard is that it completely ignored connectionless services and connectionless protocols, even though most local area networks work that way. Subsequent addenda (known in the software world as bug fixes) corrected this problem.

Perhaps the most serious criticism is that the model is dominated by a communications mentality. The relationship of computing to communications is barely mentioned anywhere, and some of the choices made are wholly inappropriate to the way computers and software work. As an example, consider the OSI primitives, listed in Fig. 1-14. In particular, think carefully about the primitives and how one might use them in a programming language.

The CONNECT.request primitive is simple. One can imagine a library procedure, *connect*, that programs can call to establish a connection. Now think about CONNECT.indication. When a message arrives, the destination process has to be signaled. In effect, it has to get an interrupt—hardly an appropriate concept for programs written in any modern high-level language. Of course, in the lowest layer, an indication (interrupt) does occur.

If the program were expecting an incoming call, it could call a library procedure *receive* to block itself. But if this were the case, why was *receive* not the primitive instead of *indication*? *Receive* is clearly oriented toward the way computers work, whereas *indication* is equally clearly oriented toward the way telephones work. Computers are different from telephones. Telephones ring. Computers do not ring. In short, the semantic model of an interrupt-driven system is conceptually a poor idea and totally at odds with all modern ideas of structured programming. This and similar problems are discussed by Langsford (1984).

Bad Implementations

Given the enormous complexity of the model and the protocols, it will come as no surprise that the initial implementations were huge, unwieldy, and slow. Everyone who tried them got burned. It did not take long for people to associate

"OSI" with "poor quality." While the products got better in the course of time, the image stuck.

In contrast, one of the first implementations of TCP/IP was part of Berkeley UNIX® and was quite good (not to mention, free). People began using it quickly, which led to a large user community, which led to improvements, which led to an even larger community. Here the spiral was upward instead of downward.

Bad Politics

On account of the initial implementation, many people, especially in academia, thought of TCP/IP as part of UNIX, and UNIX in the 1980s in academia was not unlike parenthood (then incorrectly called motherhood) and apple pie.

OSI, on the other hand, was thought to be the creature of the European telecommunication ministries, the European Community, and later the U.S. Government. This belief was only partly true, but the very idea of a bunch of government bureaucrats trying to shove a technically inferior standard down the throats of the poor researchers and programmers down in the trenches actually developing computer networks did not help much. Some people viewed this development in the same light as IBM announcing in the 1960s that PL/I was the language of the future, or DoD correcting this later by announcing that it was actually Ada®.

Despite the fact that the OSI model and protocols have been less than a resounding success, there are still a few organizations interested in it, mostly European PTTs that still have a monopoly on telecommunication. Consequently a feeble effort has been made to update OSI, resulting in a revised model published in 1994. For what was changed (little) and what should have been changed (a lot), see (Day, 1995).

1.4.5. A Critique of the TCP/IP Reference Model

The TCP/IP model and protocols have their problems too. First, the model does not clearly distinguish the concepts of service, interface, and protocol. Good software engineering practice requires differentiating between the specification and the implementation, something that OSI does very carefully, and TCP/IP does not. Consequently, the TCP/IP model is not much of a guide for designing new networks using new technologies.

Second, the TCP/IP model is not at all general and is poorly suited to describing any protocol stack other than TCP/IP. Trying to describe SNA using the TCP/IP model would be nearly impossible, for example.

Third, the host-to-network layer is not really a layer at all in the normal sense that the term is used in the context of layered protocols. It is an interface (between the network and data link layers). The distinction between an interface and a layer is a crucial one and one should not be sloppy about it.

Fourth, the TCP/IP model does not distinguish (or even mention) the physical and data link layers. These are completely different. The physical layer has to do with the transmission characteristics of copper wire, fiber optics, and wireless communication. The data link layer's job is to delimit the start and end of frames and get them from one side to the other with the desired degree of reliability. A proper model should include both as separate layers. The TCP/IP model does not do this.

Finally, although the IP and TCP protocols were carefully thought out, and well implemented, many of the other protocols were ad hoc, generally produced by a couple of graduate students hacking away until they got tired. The protocol implementations were then distributed free, which resulted in their becoming widely used, deeply entrenched, and thus hard to replace. Some of them are a bit of an embarrassment now. The virtual terminal protocol, TELNET, for example, was designed for a ten-character per second mechanical Teletype terminal. It knows nothing of graphical user interfaces and mice. Nevertheless, 25 years later, it is still in widespread use.

In summary, despite its problems, the OSI *model* (minus the session and presentation layers) has proven to be exceptionally useful for discussing computer networks. In contrast, the OSI *protocols* have not become popular. The reverse is true of TCP/IP: the *model* is practically nonexistent, but the *protocols* are widely used. Since computer scientists like to have their cake and eat it, too, in this book we will use a modified OSI model but concentrate primarily on the TCP/IP and related protocols, as well as newer ones such as SMDS, frame relay, SONET, and ATM. In effect, we will use the hybrid model of Fig. 1-21 as the framework for this book.

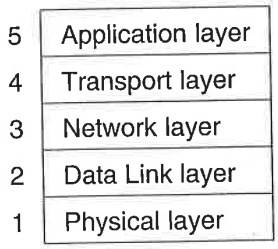


Fig. 1-21. The hybrid reference model to be used in this book.

1.5. EXAMPLE NETWORKS

Numerous networks are currently operating around the world. Some of these are public networks run by common carriers or PTTs, others are research networks, yet others are cooperative networks run by their users, and still others are commercial or corporate networks. In the following sections we will take a look

at a few current and historical networks to get an idea of what they are (or were) like and how they differ from one another.

Networks differ in their history, administration, facilities offered, technical design, and user communities. The history and administration can vary from a network carefully planned by a single organization with a well-defined goal, to an ad hoc collection of machines that have been connected to one another over the years without any master plan or central administration at all. The facilities available range from arbitrary process-to-process communication to electronic mail, file transfer, remote login, and remote execution. The technical designs can differ in the transmission media used, the naming and routing algorithms employed, the number and contents of the layers present, and the protocols used. Finally, the user community can vary from a single corporation to all the academic computer scientists in the industrialized world.

In the following sections we will look at a few examples. These are the popular commercial LAN networking package, Novell NetWare®, the worldwide Internet (including its predecessors, the ARPANET and NSFNET), and the first gigabit networks.

1.5.1. Novell NetWare

The most popular network system in the PC world is **Novell NetWare**. It was designed to be used by companies downsizing from a mainframe to a network of PCs. In such systems, each user has a desktop PC functioning as a client. In addition, some number of powerful PCs operate as servers, providing file services, database services, and other services to a collection of clients. In other words, Novell NetWare is based on the client-server model.

NetWare uses a proprietary protocol stack illustrated in Fig. 1-22. It is based on the old Xerox Network System, XNS™ but with various modifications. Novell NetWare predates OSI and is not based on it. If anything, it looks more like TCP/IP than like OSI.

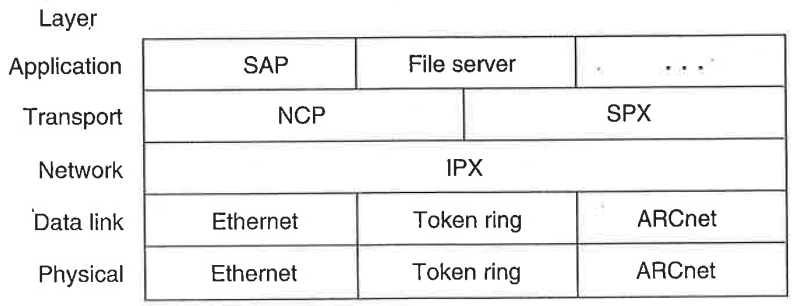


Fig. 1-22. The Novell NetWare reference model.

The physical and data link layers can be chosen from among various industry standards, including Ethernet, IBM token ring, and ARCnet. The network layer

runs an unreliable connectionless internetwork protocol called **IPX**. It passes packets transparently from source to destination, even if the source and destination are on different networks. IPX is functionally similar to IP, except that it uses 10-byte addresses instead of 4-byte addresses. The wisdom of this choice will become apparent in Chap. 5.

Above IPX comes a connection-oriented transport protocol called **NCP (Network Core Protocol)**. NCP also provides various other services besides user data transport and is really the heart of NetWare. A second protocol, **SPX**, is also available, but provides only transport. TCP is another option. Applications can choose any of them. The file system uses NCP and Lotus Notes® uses SPX, for example. The session and presentation layers do not exist. Various application protocols are present in the application layer.

As in TCP/IP, the key to the entire architecture is the internet datagram packet on top of which everything else is built. The format of an IPX packet is shown in Fig. 1-23. The *Checksum* field is rarely used, since the underlying data link layer also provides a checksum. The *Packet length* field tells how long the entire packet is, header plus data. The *Transport control* field counts how many networks the packet has traversed. When this exceeds a maximum, the packet is discarded. The *Packet type* field is used to mark various control packets. The two addresses each contain a 32-bit network number, a 48-bit machine number (the 802 LAN address), and 16-bit local address (socket) on that machine. Finally, we have the data, which occupy the rest of the packet, with the maximum size being determined by the underlying network.

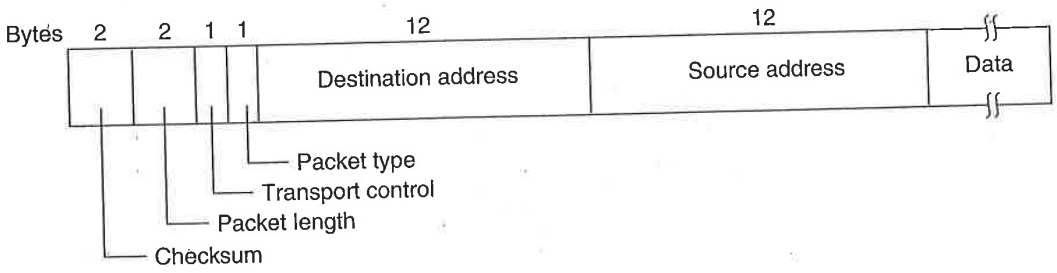


Fig. 1-23. A Novell NetWare IPX packet.

About once a minute, each server broadcasts a packet giving its address and telling what services it offers. These broadcasts use the **SAP (Service Advertising Protocol)** protocol. The packets are seen and collected by special agent processes running on the router machines. The agents use the information contained in them to construct databases of which servers are running where.

When a client machine is booted, it broadcasts a request asking where the nearest server is. The agent on the local router machine sees this request, looks in its database of servers, and matches up the request with the best server. The choice of server to use is then sent back to the client. The client can now establish

an NCP connection with the server. Using this connection, the client and server negotiate the maximum packet size. From this point on, the client can access the file system and other services using this connection. It can also query the server's database to look for other (more distant) servers.

1.5.2. The ARPANET

Let us now switch gears from LANs to WANs. In the mid-1960s, at the height of the Cold War, the DoD wanted a command and control network that could survive a nuclear war. Traditional circuit-switched telephone networks were considered too vulnerable, since the loss of one line or switch would certainly terminate all conversations using them and might even partition the network. To solve this problem, DoD turned to its research arm, ARPA (later DARPA, now ARPA again), the (periodically Defense) Advanced Research Projects Agency.

ARPA was created in response to the Soviet Union's launching Sputnik in 1957 and had the mission of advancing technology that might be useful to the military. ARPA had no scientists or laboratories, in fact, it had nothing more than an office and a small (by Pentagon standards) budget. It did its work by issuing grants and contracts to universities and companies whose ideas looked promising to it.

Several early grants went to universities for investigating the then-radical idea of packet switching, something that had been suggested by Paul Baran in a series of RAND Corporation reports published in the early 1960s. After some discussions with various experts, ARPA decided that the network the DoD needed should be a packet-switched network, consisting of a subnet and host computers.

The subnet would consist of minicomputers called **IMP**s (**Interface Message Processors**) connected by transmission lines. For high reliability, each IMP would be connected to at least two other IMPs. The subnet was to be a datagram subnet, so if some lines and IMPs were destroyed, messages could be automatically rerouted along alternative paths.

Each node of the network was to consist of an IMP and a host, in the same room, connected by a short wire. A host could send messages of up to 8063 bits to its IMP, which would then break these up into packets of at most 1008 bits and forward them independently toward the destination. Each packet was received in its entirety before being forwarded, so the subnet was the first electronic store-and-forward packet-switching network.

ARPA then put out a tender for building the subnet. Twelve companies bid for it. After evaluating all the proposals, ARPA selected BBN, a consulting firm in Cambridge, Massachusetts, and in December 1968, awarded it a contract to build the subnet and write the subnet software. BBN chose to use specially modified Honeywell DDP-316 minicomputers with 12K 16-bit words of core memory

usage must be below a predetermined level. In return, the carrier charges much less for a virtual line than a physical one.

In addition to competing with leased lines, frame relay also competes with X.25 permanent virtual circuits, except that it operates at higher speeds, usually 1.5 Mbps, and provides fewer features.

Frame relay provides a minimal service, primarily a way to determine the start and end of each frame, and detection of transmission errors. If a bad frame is received, the frame relay service simply discards it. It is up to the user to discover that a frame is missing and take the necessary action to recover. Unlike X.25, frame relay does not provide acknowledgements or normal flow control. It does have a bit in the header, however, which one end of a connection can set to indicate to the other end that problems exist. The use of this bit is up to the users.

1.6.4. Broadband ISDN and ATM

Even if the above services become popular, the telephone companies are still faced with a far more fundamental problem: multiple networks. POTS (Plain Old Telephone Service) and Telex use the old circuit-switched network. Each of the new data services such as SMDS and frame relay uses its own packet-switching network. DQDB is different from these, and the internal telephone company call management network (SSN 7) is yet another network. Maintaining all these separate networks is a major headache, and there is another network, cable television, that the telephone companies do not control and would like to.

The perceived solution is to invent a single new network for the future that will replace the entire telephone system and all the specialized networks with a single integrated network for all kinds of information transfer. This new network will have a huge data rate compared to all existing networks and services and will make it possible to offer a large variety of new services. This is not a small project, and it is certainly not going to happen overnight, but it is now under way.

The new wide area service is called **B-ISDN (Broadband Integrated Services Digital Network)**. It will offer video on demand, live television from many sources, full motion multimedia electronic mail, CD-quality music, LAN interconnection, high-speed data transport for science and industry and many other services that have not yet even been thought of, all over the telephone line.

The underlying technology that makes B-ISDN possible is called **ATM (Asynchronous Transfer Mode)** because it is not synchronous (tied to a master clock), as most long distance telephone lines are. Note that the acronym ATM here has nothing to do with the Automated Teller Machines many banks provide (although an ATM machine can use an ATM network to talk to its bank).

A great deal of work has already been done on ATM and on the B-ISDN system that uses it, although there is more ahead. For more information on this subject, see (Fischer et al., 1994; Gasman, 1994; Goralski, 1995; Kim et al., 1994; Kyas, 1995; McDysan and Spohn, 1995; and Stallings, 1995a).

The basic idea behind ATM is to transmit all information in small, fixed-size packets called **cells**. The cells are 53 bytes long, of which 5 bytes are header and 48 bytes are payload, as shown in Fig. 1-29. ATM is both a technology (hidden from the users) and potentially a service (visible to the users). Sometimes the service is called **cell relay**, as an analogy to frame relay.

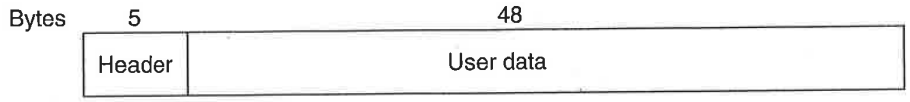


Fig. 1-29. An ATM cell.

The use of a cell-switching technology is a gigantic break with the 100-year old tradition of circuit switching (establishing a copper path) within the telephone system. There are a variety of reasons why cell switching was chosen, among them are the following. First, cell switching is highly flexible and can handle both constant rate traffic (audio, video) and variable rate traffic (data) easily. Second, at the very high speeds envisioned (gigabits per second are within reach), digital switching of cells is easier than using traditional multiplexing techniques, especially using fiber optics. Third, for television distribution, broadcasting is essential; cell switching can provide this and circuit switching cannot.

ATM networks are connection-oriented. Making a call requires first sending a message to set up the connection. After that, subsequent cells all follow the same path to the destination. Cell delivery is not guaranteed, but their order is. If cells 1 and 2 are sent in that order, then if both arrive, they will arrive in that order, never first 2 then 1.

ATM networks are organized like traditional WANs, with lines and switches (routers). The intended speeds for ATM networks are 155 Mbps and 622 Mbps, with the possibility of gigabit speeds later. The 155-Mbps speed was chosen because this is about what is needed to transmit high definition television. The exact choice of 155.52 Mbps was made for compatibility with AT&T's SONET transmission system. The 622 Mbps speed was chosen so four 155-Mbps channels could be sent over it. By now it should be clear why some of the gigabit testbeds operated at 622 Mbps: they used ATM.

When ATM was proposed, virtually all the discussion (i.e., the hype) was about video on demand to every home and replacing the telephone system, as described above. Since then, other developments have become important. Many organizations have run out of bandwidth on their campus or building-wide LANs and are being forced to go to some kind of switched system that has more bandwidth than does a single LAN. Also, in client-server computing, some applications need the ability to talk to certain servers at high speed. ATM is certainly a major candidate for both of these applications. Nevertheless, it is a bit of a letdown to go from a goal of trying to replace the entire low-speed analog telephone

system with a high-speed digital one to a goal of trying connect all the Ethernets on campus. LAN interconnection using ATM is discussed in (Kavak, 1995; Newman, 1994; and Truong et al., 1995).

It is also worth pointing out that different organizations involved in ATM have different (financial) interests. The long-distance telephone carriers and PTTs are mostly interested in using ATM to upgrade the telephone system and compete with the cable TV companies in electronic video distribution. The computer vendors see campus ATM LANs as the big moneymaker (for them). All these competing interests do not make the ongoing standardization process any easier, faster, or more coherent. Also, politics and power within the organization standardizing ATM (The ATM Forum) have considerable influence on where ATM is going.

The B-ISDN ATM Reference Model

Let us now turn back to the technology of ATM, especially as used in the (future) telephone system. Broadband ISDN using ATM has its own reference model, different from the OSI model and also different from the TCP/IP model. This model is shown in Fig. 1-30. It consists of three layers, the physical, ATM, and ATM adaptation layers, plus whatever the users want to put on top of that.

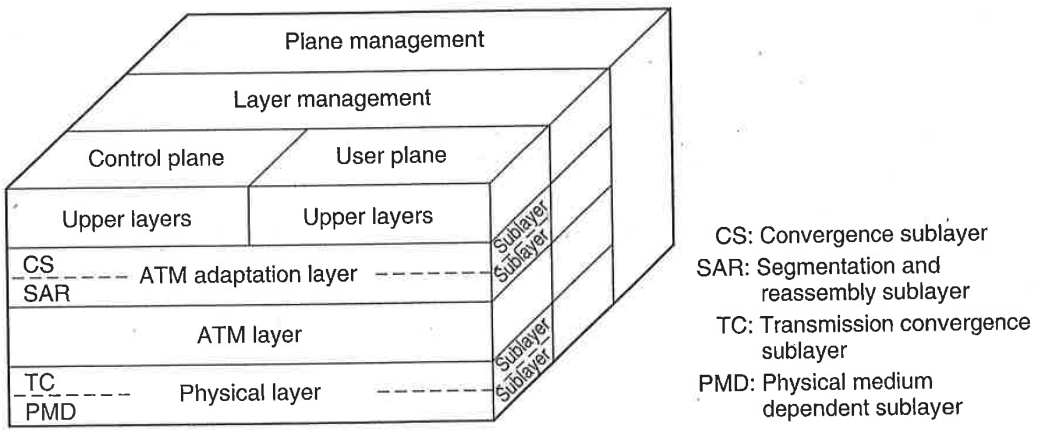


Fig. 1-30. The B-ISDN ATM reference model.

The physical layer deals with the physical medium: voltages, bit timing, and various other issues. ATM does not prescribe a particular set of rules, but instead says that ATM cells may be sent on a wire or fiber by themselves, but they may also be packaged inside the payload of other carrier systems. In other words, ATM has been designed to be independent of the transmission medium.

The **ATM layer** deals with cells and cell transport. It defines the layout of a cell and tells what the header fields mean. It also deals with establishment and release of virtual circuits. Congestion control is also located here.

Because most applications do not want to work directly with cells (although some may), a layer above the ATM layer has been defined that allows users to send packets larger than a cell. The ATM interface segments these packets, transmits the cells individually, and reassembles them at the other end. This layer is the **AAL (ATM Adaptation Layer)**.

Unlike the earlier two-dimensional reference models, the ATM model is defined as being three-dimensional, as shown in Fig. 1-30. The **user plane** deals with data transport, flow control, error correction, and other user functions. In contrast, the **control plane** is concerned with connection management. The layer and plane management functions relate to resource management and interlayer coordination.

The physical and AAL layers are each divided into two sublayers, one at the bottom that does the work and a convergence sublayer on top that provides the proper interface to the layer above it. The functions of the layers and sublayers are given in Fig. 1-31.

OSI layer	ATM layer	ATM sublayer	Functionality
3/4	AAL	CS	Providing the standard interface (convergence)
		SAR	Segmentation and reassembly
2/3	ATM		Flow control Cell header generation/extraction Virtual circuit/path management Cell multiplexing/demultiplexing
2	Physical	TC	Cell rate decoupling Header checksum generation and verification Cell generation Packing/unpacking cells from the enclosing envelope Frame generation
1		PMD	Bit timing Physical network access

Fig. 1-31. The ATM layers and sublayers, and their functions.

The **PMD (Physical Medium Dependent)** sublayer interfaces to the actual cable. It moves the bits on and off and handles the bit timing. For different carriers and cables, this layer will be different.

The other sublayer of the physical layer is the **TC (Transmission Convergence)** sublayer. When cells are transmitted, the TC layer sends them as a string of bits to the PMD layer. Doing this is easy. At the other end, the TC sublayer gets a pure incoming bit stream from the PMD sublayer. Its job is to convert this

bit stream into a cell stream for the ATM layer. It handles all the issues related to telling where cells begin and end in the bit stream. In the ATM model, this functionality is in the physical layer. In the OSI model and in pretty much all other networks, the job of framing, that is, turning a raw bit stream into a sequence of frames or cells, is the data link layer's task. For that reason we will discuss it in this book along with the data link layer, not with the physical layer.

As we mentioned earlier, the ATM layer manages cells, including their generation and transport. Most of the interesting aspects of ATM are located here. It is a mixture of the OSI data link and network layers, but it is not split into sublayers.

The AAL layer is split into a **SAR** (**Segmentation And Reassembly**) sublayer and a **CS** (**Convergence Sublayer**). The lower sublayer breaks packets up into cells on the transmission side and puts them back together again at the destination. The upper sublayer makes it possible to have ATM systems offer different kinds of services to different applications (e.g., file transfer and video on demand have different requirements concerning error handling, timing, etc.).

Perspective on ATM

To a considerable extent, ATM is a project invented by the telephone industry because after Ethernet was widely installed, the computer industry never rallied around any higher-speed network technology to make it standard. The telephone companies filled this vacuum with ATM, although in October 1991, many computer vendors joined with the telephone companies to set up the **ATM Forum**, an industry group that will guide the future of ATM.

Although ATM promises the ability to deliver information anywhere at speeds soon to exceed 1 Gbps, delivering on this promise will not be easy. ATM is basically high-speed packet-switching, a technology the telephone companies have little experience with. What they do have, is a massive investment in a different technology (circuit switching) that is in concept unchanged since the days of Alexander Graham Bell. Needless to say, this transition will not happen quickly, all the more so because it is a revolutionary change rather than an evolutionary one, and revolutions never go smoothly.

The economics of installing ATM worldwide also have to be considered. A substantial fraction of the existing telephone system will have to be replaced. Who will pay for this? How much will consumers be willing to pay to get a movie on demand electronically, when they can get one at the local video store for a couple of dollars? Finally, the question of where many of the advanced services are provided is crucial. If they are provided by the network, the telephone companies will profit from them. If they are provided by computers attached to the network, the manufacturers and operators of these devices make the profits. The users may not care, but the telephone companies and computer vendors certainly do, and this will surely affect their interest in making ATM happen.

1.6.5. Comparison of Services

The reader may be wondering why so many incompatible and overlapping services exist, including DQDB, SMDS, X.25, frame relay, ATM, and more. The underlying reason is the 1984 decision to break up AT&T and foster competition in the telecommunications industry. Different companies with different interests and technologies are now free to offer whatever services they think there is a demand for, and many of them are doing this with a vengeance.

To recap some of the services we have touched on in this chapter, DQDB is an unswitched MAN technology that allows 53-byte cells (of which 44 are payload) to be sent down long wires within a city. SMDS is a switched datagram technology for sending datagrams anywhere in a network at 45 Mbps. X.25 is an older connection-oriented networking technology for transmitting small variable-sized packets at 64 kbps. Frame relay is a service that provides virtual leased lines at speeds around 1.5 Mbps. Finally, ATM is designed to replace the entire circuit-switched telephone system with cell switching and be able to handle data and television as well. Some differences between these competitors are summarized in Fig. 1-32.

Issue	DQDB	SMDS	X.25	Frame Relay	ATM AAL
Connection oriented	Yes	No	Yes	Yes	Yes
Normal speed (Mbps)	45	45	.064	1.5	155
Switched	No	Yes	Yes	No	Yes
Fixed-size payload	Yes	No	No	No	No
Max payload	44	9188	128	1600	Variable
Permanent VCs	No	No	Yes	Yes	Yes
Multicasting	No	Yes	No	No	Yes

Fig. 1-32. Different networking services.

1.7. NETWORK STANDARDIZATION

Many network vendors and suppliers exist, each with their own ideas of how things should be done. Without coordination, there would be complete chaos, and users would be able to get nothing done. The only way out is to agree upon some network standards.

Not only do standards allow different computers to communicate, but they also increase the market for products adhering to the standard, which leads to

4

THE MEDIUM ACCESS SUBLAYER

As we pointed out in Chap. 1, networks can be divided into two categories: those using point-to-point connections and those using broadcast channels. This chapter deals with broadcast networks and their protocols.

In any broadcast network, the key issue is how to determine who gets to use the channel when there is competition for it. To make this point clearer, consider a conference call in which six people, on six different telephones, are all connected together so that each one can hear and talk to all the others. It is very likely that when one of them stops speaking, two or more will start talking at once, leading to chaos. In a face-to-face meeting, chaos is avoided by external means, for example, at a meeting, people raise their hands to request permission to speak. When only a single channel is available, determining who should go next is much harder. Many protocols for solving the problem are known and form the contents of this chapter. In the literature, broadcast channels are sometimes referred to as **multiaccess channels** or **random access channels**.

The protocols used to determine who goes next on a multiaccess channel belong to a sublayer of the data link layer called the **MAC (Medium Access Control)** sublayer. The MAC sublayer is especially important in LANs, nearly all of which use a multiaccess channel as the basis of their communication. WANs, in contrast, use point-to-point links, except for satellite networks. Because multiaccess channels and LANs are so closely related, in this chapter we will discuss LANs in general, as well as satellite and some other broadcast networks.

Technically, the MAC sublayer is the bottom part of the data link layer, so logically we should have studied it before examining all the point-to-point protocols in Chap. 3. Nevertheless, for most people, understanding protocols involving multiple parties is easier after two-party protocols are well understood. For that reason we have deviated slightly from a strict bottom-up order of presentation.

4.1. THE CHANNEL ALLOCATION PROBLEM

The central theme of this chapter is how to allocate a single broadcast channel among competing users. We will first look at static and dynamic schemes in general. Then we will examine a number of specific algorithms.

4.1.1. Static Channel Allocation in LANs and MANs

The traditional way of allocating a single channel, such as a telephone trunk, among multiple competing users is Frequency Division Multiplexing (FDM). If there are N users, the bandwidth is divided into N equal sized portions (see Fig. 2-24), each user being assigned one portion. Since each user has a private frequency band, there is no interference between users. When there is only a small and fixed number of users, each of which has a heavy (buffered) load of traffic (e.g., carriers' switching offices), FDM is a simple and efficient allocation mechanism.

However, when the number of senders is large and continuously varying, or the traffic is bursty, FDM presents some problems. If the spectrum is cut up into N regions, and fewer than N users are currently interested in communicating, a large piece of valuable spectrum will be wasted. If more than N users want to communicate, some of them will be denied permission, for lack of bandwidth, even if some of the users who have been assigned a frequency band hardly ever transmit or receive anything.

However, even assuming that the number of users could somehow be held constant at N, dividing the single available channel into static subchannels is inherently inefficient. The basic problem is that when some users are quiescent, their bandwidth is simply lost. They are not using it, and no one else is allowed to use it either. Furthermore, in most computer systems, data traffic is extremely bursty (peak traffic to mean traffic ratios of 1000:1 are common). Consequently, most of the channels will be idle most of the time.

The poor performance of static FDM can easily be seen from a simple queueing theory calculation. Let us start with the mean time delay, T, for a channel of capacity C bps, with an arrival rate of λ frames/sec, each frame having a length drawn from an exponential probability density function with mean $1/\mu$ bits/frame:

$$T = \frac{1}{\mu C - \lambda}$$

Now let us divide the single channel up into N independent subchannels, each

4.3.1. IEEE Standard 802.3 and Ethernet

The IEEE 802.3 standard is for a 1-persistent CSMA/CD LAN. To review the idea, when a station wants to transmit, it listens to the cable. If the cable is busy, the station waits until it goes idle; otherwise it transmits immediately. If two or more stations simultaneously begin transmitting on an idle cable, they will collide. All colliding stations then terminate their transmission, wait a random time, and repeat the whole process all over again.

The 802.3 standard has an interesting history. The real beginning was the ALOHA system constructed to allow radio communication between machines scattered over the Hawaiian Islands. Later, carrier sensing was added, and Xerox PARC built a 2.94-Mbps CSMA/CD system to connect over 100 personal workstations on a 1-km cable (Metcalfe and Boggs, 1976). This system was called **Ethernet** after the *luminiferous ether*, through which electromagnetic radiation was once thought to propagate. (When the Nineteenth Century British physicist James Clerk Maxwell discovered that electromagnetic radiation could be described by a wave equation, scientists assumed that space must be filled with some ethereal medium in which the radiation was propagating. Only after the famous Michelson-Morley experiment in 1887, did physicists discover that electromagnetic radiation could propagate in a vacuum.)

The Xerox Ethernet was so successful that Xerox, DEC, and Intel drew up a standard for a 10-Mbps Ethernet. This standard formed the basis for 802.3. The published 802.3 standard differs from the Ethernet specification in that it describes a whole family of 1-persistent CSMA/CD systems, running at speeds from 1 to 10-Mbps on various media. Also, the one header field differs between the two (the 802.3 length field is used for packet type in Ethernet). The initial standard also gives the parameters for a 10 Mbps baseband system using 50-ohm coaxial cable. Parameter sets for other media and speeds came later.

Many people (incorrectly) use the name "Ethernet" in a generic sense to refer to all CSMA/CD protocols, even though it really refers to a specific product that almost implements 802.3. We will use the terms "802.3" and "CSMA/CD" except when specifically referring to the Ethernet product in the next few paragraphs.

802.3 Cabling

Since the name "Ethernet" refers to the cable (the ether), let us start our discussion there. Five types of cabling are commonly used, as shown in Fig. 4-17. Historically, **10Base5** cabling, popularly called **thick Ethernet,** came first. It resembles a yellow garden hose, with markings every 2.5 meters to show where the taps go. (The 802.3 standard does not actually *require* the cable to be yellow, but it does *suggest* it.) Connections to it are generally made using **vampire taps**, in which a pin is carefully forced halfway into the coaxial cable's core. The

notation 10Base5 means that it operates at 10 Mbps, uses baseband signaling, and can support segments of up to 500 meters.

Name	Cable	Max. segment	Nodes/seg.	Advantages
10Base5	Thick coax	500 m	100	Good for backbones
10Base2	Thin coax	200 m	30	Cheapest system
10Base-T	Twisted pair	100 m	1024	Easy maintenance
10Base-F	Fiber optics	2000 m	1024	Best between buildings

Fig. 4-17. The most common kinds of baseband 802.3 LANs.

Historically, the second cable type was **10Base2** or **thin Ethernet,** which, in contrast to the garden-hose-like thick Ethernet, bends easily. Connections to it are made using industry standard BNC connectors to form T junctions, rather than using vampire taps. These are easier to use and more reliable. Thin Ethernet is much cheaper and easier to install, but it can run for only 200 meters and can handle only 30 machines per cable segment.

Detecting cable breaks, bad taps, or loose connectors can be a major problem with both media. For this reason, techniques have been developed to track them down. Basically, a pulse of known shape is injected into the cable. If the pulse hits an obstacle or the end of the cable, an echo will be generated and sent back. By carefully timing the interval between sending the pulse and receiving the echo, it is possible to localize the origin of the echo. This technique is called **time domain reflectometry**.

The problems associated with finding cable breaks have driven systems toward a different kind of wiring pattern, in which all stations have a cable running to a central **hub**. Usually, these wires are telephone company twisted pairs, since most office buildings are already wired this way, and there are normally plenty of spare pairs available. This scheme is called **10Base-T**.

These three wiring schemes are illustrated in Fig. 4-18. For 10Base5, a **transceiver** is clamped securely around the cable so that its tap makes contact with the inner core. The transceiver contains the electronics that handle carrier detection and collision detection. When a collision is detected, the transceiver also puts a special invalid signal on the cable to ensure that all other transceivers also realize that a collision has occurred.

With 10Base5, a **transceiver cable** connects the transceiver to an interface board in the computer. The transceiver cable may be up to 50 meters long and contains five individually shielded twisted pairs. Two of the pairs are for data in and data out, respectively. Two more are for control signals in and out. The fifth pair, which is not always used, allows the computer to power the transceiver electronics. Some transceivers allow up to eight nearby computers to be attached to them, to reduce the number of transceivers needed.

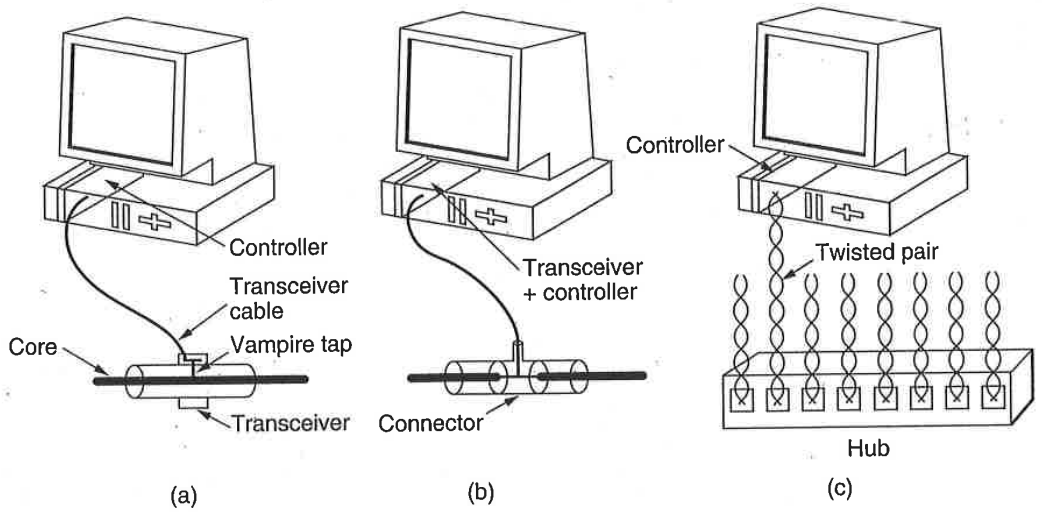


Fig. 4-18. Three kinds of 802.3 cabling. (a) 10Base5. (b) 10Base2. (c) 10Base-T.

The transceiver cable terminates on an interface board inside the computer. The interface board contains a controller chip that transmits frames to, and receives frames from, the transceiver. The controller is responsible for assembling the data into the proper frame format, as well as computing checksums on outgoing frames and verifying them on incoming frames. Some controller chips also manage a pool of buffers for incoming frames, a queue of buffers to be transmitted, DMA transfers with the host computers, and other aspects of network management.

With 10Base2, the connection to the cable is just a passive BNC T-junction connector. The transceiver electronics are on the controller board, and each station always has its own transceiver.

With 10Base-T, there is no cable at all, just the hub (a box full of electronics). Adding or removing a station is simpler in this configuration, and cable breaks can be detected easily. The disadvantage of 10Base-T is that the maximum cable run from the hub is only 100 meters, maybe 150 meters if high-quality (category 5) twisted pairs are used. Also, a large hub costs thousands of dollars. Still, 10Base-T is becoming steadily more popular due to the ease of maintenance that it offers. A faster version of 10Base-T (100Base-T) will be discussed later in this chapter.

A fourth cabling option for 802.3 is **10Base-F**, which uses fiber optics. This alternative is expensive due to the cost of the connectors and terminators, but it has excellent noise immunity and is the method of choice when running between buildings or widely separated hubs.

Figure 4-19 shows different ways of wiring up a building. In Fig. 4-19(a), a single cable is snaked from room to room, with each station tapping onto it at the nearest point. In Fig. 4-19(b), a vertical spine runs from the basement to the roof,

with horizontal cables on each floor connected to it by special amplifiers (repeaters). In some buildings the horizontal cables are thin, and the backbone is thick. The most general topology is the tree, as in Fig. 4-19(c), because a network with two paths between some pairs of stations would suffer from interference between the two signals.

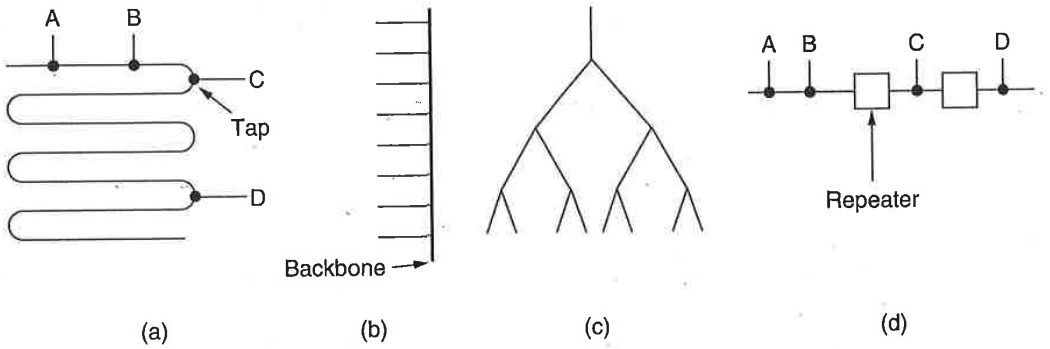


Fig. 4-19. Cable topologies. (a) Linear. (b) Spine. (c) Tree. (d) Segmented.

Each version of 802.3 has a maximum cable length per segment. To allow larger networks, multiple cables can be connected by **repeaters**, as shown in Fig. 4-19(d). A repeater is a physical layer device. It receives, amplifies, and retransmits signals in both directions. As far as the software is concerned, a series of cable segments connected by repeaters is no different than a single cable (except for some delay introduced by the repeaters). A system may contain multiple cable segments and multiple repeaters, but no two transceivers may be more than 2.5 km apart and no path between any two transceivers may traverse more than four repeaters.

Manchester Encoding

None of the versions of 802.3 use straight binary encoding with 0 volts for a 0 bit and 5 volts for a 1 bit because it leads to ambiguities. If one station sends the bit string 0001000, others might falsely interpret it as 10000000 or 01000000 because they cannot tell the difference between an idle sender (0 volts) and a 0 bit (0 volts).

What is needed is a way for receivers to unambiguously determine the start, end, or middle of each bit without reference to an external clock. Two such approaches are called **Manchester encoding** and **differential Manchester encoding**. With Manchester encoding, each bit period is divided into two equal intervals. A binary 1 bit is sent by having the voltage set high during the first interval and low in the second one. A binary 0 is just the reverse: first low and then high. This scheme ensures that every bit period has a transition in the middle, making it easy for the receiver to synchronize with the sender. A

disadvantage of Manchester encoding is that it requires twice as much bandwidth as straight binary encoding, because the pulses are half the width. Manchester encoding is shown in Fig. 4-20(b).

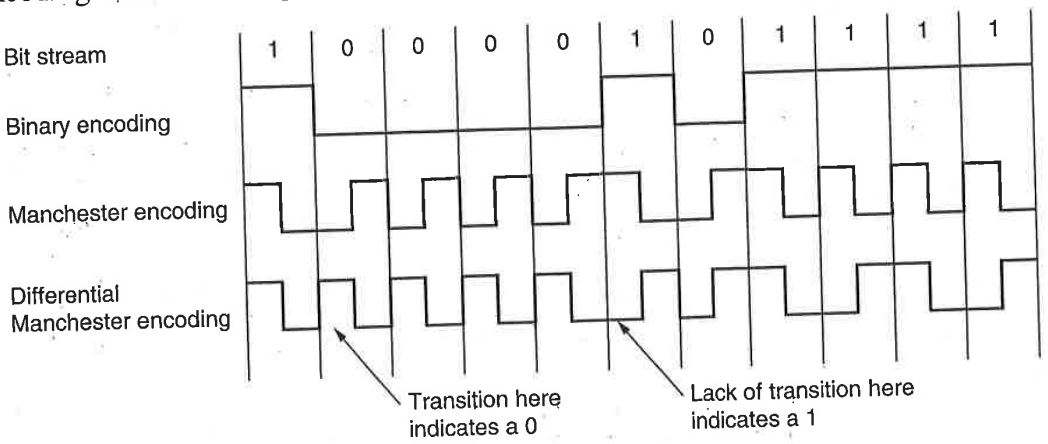


Fig. 4-20. (a) Binary encoding. (b) Manchester encoding. (c) Differential Manchester encoding.

Differential Manchester encoding, shown in Fig. 4-20(c), is a variation of basic Manchester encoding. In it, a 1 bit is indicated by the absence of a transition at the start of the interval. A 0 bit is indicated by the presence of a transition at the start of the interval. In both cases, there is a transition in the middle as well. The differential scheme requires more complex equipment but offers better noise immunity. All 802.3 baseband systems use Manchester encoding due to its simplicity. The high signal is +0.85 volts and the low signal is −0.85 volts, giving a DC value of 0 volts.

The 802.3 MAC Sublayer Protocol

The 802.3 (IEEE, 1985a) frame structure is shown in Fig. 4-21. Each frame starts with a *Preamble* of 7 bytes, each containing the bit pattern 10101010. The Manchester encoding of this pattern produces a 10-MHz square wave for 5.6 μsec to allow the receiver's clock to synchronize with the sender's. Next comes a *Start of frame* byte containing 10101011 to denote the start of the frame itself.

The frame contains two addresses, one for the destination and one for the source. The standard allows 2-byte and 6-byte addresses, but the parameters defined for the 10-Mbps baseband standard use only the 6-byte addresses. The high-order bit of the destination address is a 0 for ordinary addresses and 1 for group addresses. Group addresses allow multiple stations to listen to a single address. When a frame is sent to a group address, all the stations in the group receive it. Sending to a group of stations is called **multicast**. The address consisting of all 1 bits is reserved for **broadcast**. A frame containing all 1s in the destination field is delivered to all stations on the network.

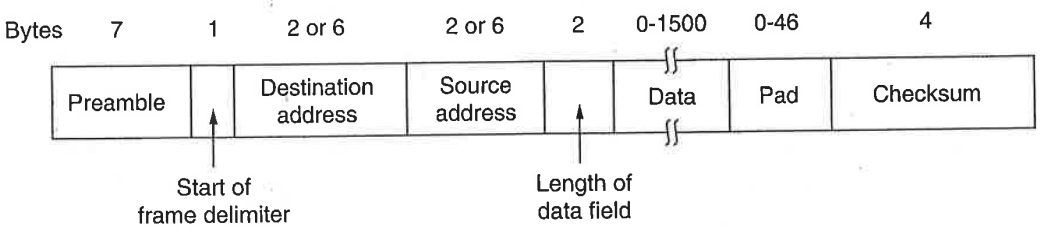


Fig. 4-21. The 802.3 frame format.

Another interesting feature of the addressing is the use of bit 46 (adjacent to the high-order bit) to distinguish local from global addresses. Local addresses are assigned by each network administrator and have no significance outside the local network. Global addresses, in contrast, are assigned by IEEE to ensure that no two stations anywhere in the world have the same global address. With $48 - 2 = 46$ bits available, there are about 7×10^{13} global addresses. The idea is that any station can uniquely address any other station by just giving the right 48-bit number. It is up to the network layer to figure out how to locate the destination.

The *Length* field tells how many bytes are present in the data field, from a minimum of 0 to a maximum of 1500. While a data field of 0 bytes is legal, it causes a problem. When a transceiver detects a collision, it truncates the current frame, which means that stray bits and pieces of frames appear on the cable all the time. To make it easier to distinguish valid frames from garbage, 802.3 states that valid frames must be at least 64 bytes long, from destination address to checksum. If the data portion of a frame is less than 46 bytes, the pad field is used to fill out the frame to the minimum size.

Another (and more important) reason for having a minimum length frame is to prevent a station from completing the transmission of a short frame before the first bit has even reached the far end of the cable, where it may collide with another frame. This problem is illustrated in Fig. 4-22. At time 0, station *A*, at one end of the network, sends off a frame. Let us call the propagation time for this frame to reach the other end τ. Just before the frame gets to the other end (i.e., at time $\tau - \varepsilon$) the most distant station, *B*, starts transmitting. When *B* detects that it is receiving more power than it is putting out, it knows that a collision has occurred, so it aborts its transmission and generates a 48-bit noise burst to warn all other stations. At about time 2τ, the sender sees the noise burst and aborts its transmission, too. It then waits a random time before trying again.

If a station tries to transmit a very short frame, it is conceivable that a collision occurs, but the transmission completes before the noise burst gets back at 2τ. The sender will then incorrectly conclude that the frame was successfully sent. To prevent this situation from occurring, all frames must take more than 2τ to send. For a 10-Mbps LAN with a maximum length of 2500 meters and four

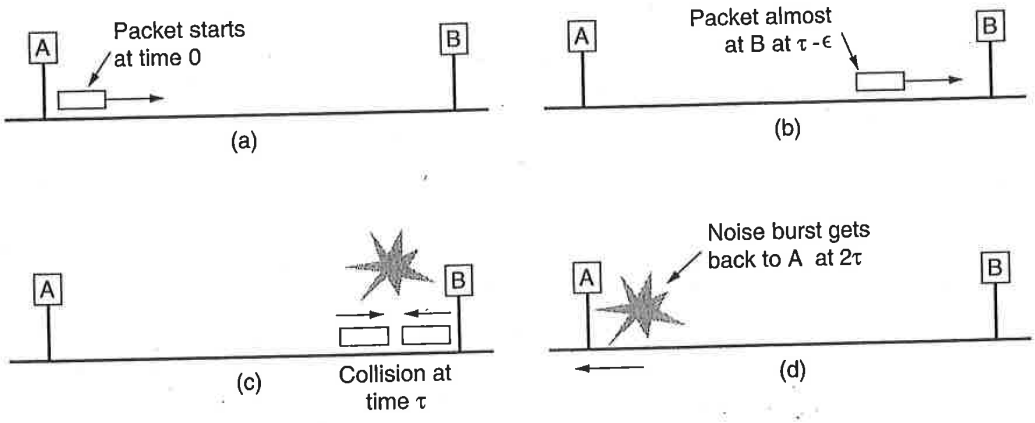


Fig. 4-22. Collision detection can take as long as 2τ.

repeaters (from the 802.3 specification), the minimum allowed frame must take 51.2 μsec. This time corresponds to 64 bytes. Frames with fewer bytes are padded out to 64 bytes.

As the network speed goes up, the minimum frame length must go up or the maximum cable length must come down, proportionally. For a 2500-meter LAN operating at 1 Gbps, the minimum frame size would have to be 6400 bytes. Alternatively, the minimum frame size could be 640 bytes and the maximum distance between any two stations 250 meters. These restrictions are becoming increasingly painful as we move toward gigabit networks.

The final 802.3 field is the *Checksum*. It is effectively a 32-bit hash code of the data. If some data bits are erroneously received (due to noise on the cable), the checksum will almost certainly be wrong, and the error will be detected. The checksum algorithm is a cyclic redundancy check of the kind discussed in Chap. 3.

The Binary Exponential Backoff Algorithm

Let us now see how randomization is done when a collision occurs. The model is that of Fig. 4-5. After a collision, time is divided up into discrete slots whose length is equal to the worst case round-trip propagation time on the ether (2τ). To accommodate the longest path allowed by 802.3 (2.5 km and four repeaters), the slot time has been set to 512 bit times, or 51.2 μsec.

After the first collision, each station waits either 0 or 1 slot times before trying again. If two stations collide and each one picks the same random number, they will collide again. After the second collision, each one picks either 0, 1, 2, or 3 at random and waits that number of slot times. If a third collision occurs (the probability of this happening is 0.25), then the next time the number of slots to wait is chosen at random from the interval 0 to $2^3 - 1$.

In general, after i collisions, a random number between 0 and $2^i - 1$ is chosen, and that number of slots is skipped. However, after ten collisions have been reached, the randomization interval is frozen at a maximum of 1023 slots. After 16 collisions, the controller throws in the towel and reports failure back to the computer. Further recovery is up to higher layers.

This algorithm, called **binary exponential backoff**, was chosen to dynamically adapt to the number of stations trying to send. If the randomization interval for all collisions was 1023, the chance of two stations colliding for a second time would be negligible, but the average wait after a collision would be hundreds of slot times, introducing significant delay. On the other hand, if each station always delayed for either zero or one slots, then if 100 stations ever tried to send at once, they would collide over and over until 99 of them picked 0 and the remaining station picked 1, or vice versa. This might take years. By having the randomization interval grow exponentially as more and more consecutive collisions occur, the algorithm ensures a low delay when only a few stations collide but also ensures that the collision is resolved in a reasonable interval when many stations collide.

As described so far, CSMA/CD provides no acknowledgements. Since the mere absence of collisions does not guarantee that bits were not garbled by noise spikes on the cable, for reliable communication the destination must verify the checksum, and if correct, send back an acknowledgement frame to the source. Normally, this acknowledgement would be just another frame as far as the protocol is concerned and would have to fight for channel time just like a data frame. However, a simple modification to the contention algorithm would allow speedy confirmation of frame receipt (Tokoro and Tamaru, 1977). All that would be needed is to reserve the first contention slot following successful transmission for the destination station.

802.3 Performance

Now let us briefly examine the performance of 802.3 under conditions of heavy and constant load, that is, k stations always ready to transmit. A rigorous analysis of the binary exponential backoff algorithm is complicated. Instead we will follow Metcalfe and Boggs (1976) and assume a constant retransmission probability in each slot. If each station transmits during a contention slot with probability p, the probability A that some station acquires the channel in that slot is

$$A = kp(1-p)^{k-1} \tag{4-6}$$

A is maximized when $p = 1/k$, with $A \to 1/e$ as $k \to \infty$. The probability that the contention interval has exactly j slots in it is $A(1-A)^{j-1}$, so the mean number of slots per contention is given by

$$\sum_{j=0}^{\infty} jA(1-A)^{j-1} = \frac{1}{A}$$

Since each slot has a duration 2τ, the mean contention interval, w, is $2\tau/A$.

Assuming optimal p, the mean number of contention slots is never more than e, so w is at most $2\tau e \approx 5.4\tau$.

If the mean frame takes P sec to transmit, when many stations have frames to send,

$$\text{Channel efficiency} = \frac{P}{P + 2\tau/A} \tag{4-7}$$

Here we see where the maximum cable distance between any two stations enters into the performance figures, giving rise to topologies other than that of Fig. 4-19(a). The longer the cable, the longer the contention interval. By allowing no more than 2.5 km of cable and four repeaters between any two transceivers, the round-trip time can be bounded to 51.2 μsec, which at 10 Mbps corresponds to 512 bits or 64 bytes, the minimum frame size.

It is instructive to formulate Eq. (4-7) in terms of the frame length, F, the network bandwidth, B, the cable length, L, and the speed of signal propagation, c, for the optimal case of e contention slots per frame. With $P = F/B$, Eq. (4-7) becomes

$$\text{Channel efficiency} = \frac{1}{1 + 2BLe/cF} \tag{4-8}$$

When the second term in the denominator is large, network efficiency will be low. More specifically, increasing network bandwidth or distance (the BL product) reduces efficiency for a given frame size. Unfortunately, much research on network hardware is aimed precisely at increasing this product. People want high bandwidth over long distances (fiber optic MANs, for example), which suggests that 802.3 may not be the best system for these applications.

In Fig. 4-23, the channel efficiency is plotted versus number of ready stations for $2\tau = 51.2$ μsec and a data rate of 10 Mbps using Eq. (4-12). With a 64-byte slot time, it is not surprising that 64-byte frames are not efficient. On the other hand, with 1024-byte frames and an asymptotic value of e 64-byte slots per contention interval, the contention period is 174 bytes long and the efficiency is 0.85.

To determine the mean number of stations ready to transmit under conditions of high load, we can use the following (crude) observation. Each frame ties up the channel for one contention period and one frame transmission time, for a total of $P + w$ sec. The number of frames per second is therefore $1/(P + w)$. If each station generates frames at a mean rate of λ frames/sec, when the system is in state k the total input rate of all unblocked stations combined is $k\lambda$ frames/sec. Since in equilibrium the input and output rates must be identical, we can equate these two expressions and solve for k. (Notice that w is a function of k.) A more sophisticated analysis is given in (Bertsekas and Gallager, 1992).

It is probably worth mentioning that there has been a large amount of theoretical performance analysis of 802.3 (and other networks). Virtually all of this work has assumed that traffic is Poisson. As researchers have begun looking at real

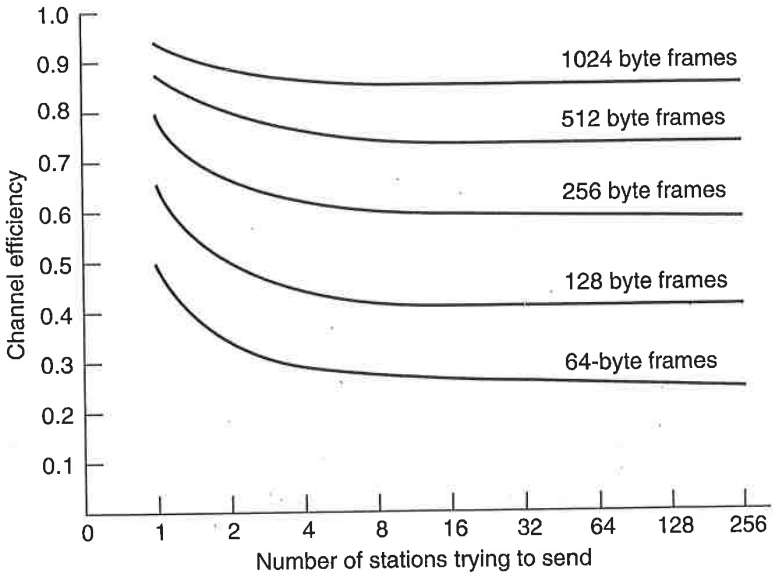


Fig. 4-23. Efficiency of 802.3 at 10 Mbps with 512-bit slot times.

data, it now appears that network traffic is rarely Poisson, but self-similar (Paxson and Floyd, 1994; and Willinger et al., 1995). What this means is that averaging over long periods of time does not smooth out the traffic. The average number of packets in each minute of an hour has as much variance as the average number of packets in each second of a minute. The consequence of this discovery is that most models of network traffic do not apply to the real world and should be taken with a grain (or better yet, a metric ton) of salt.

Switched 802.3 LANs

As more and more stations are added to an 802.3 LAN, the traffic will go up. Eventually, the LAN will saturate. One way out is to go to a higher speed, say from 10 Mbps to 100 Mbps. This solution requires throwing out all the 10 Mbps adaptor cards and buying new ones, which is expensive. If the 802.3 chips are on the computers' main circuit boards, it may not even be possible to replace them.

Fortunately, a different, less drastic solution is possible: a switched 802.3 LAN, as shown in Fig. 4-24. The heart of this system is a switch containing a high-speed backplane and room for typically 4 to 32 plug-in line cards, each containing one to eight connectors. Most often, each connector has a 10Base-T twisted pair connection to a single host computer.

When a station wants to transmit an 802.3 frame, it outputs a standard frame to the switch. The plug-in card getting the frame checks to see if it is destined for one of the other stations connected to the same card. If so, the frame is copied

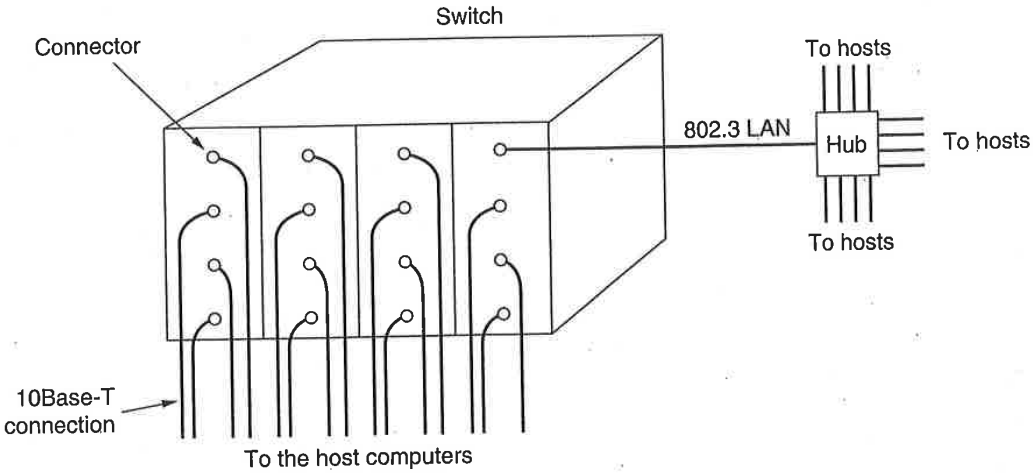


Fig. 4-24. A switched 802.3 LAN.

there. If not, the frame is sent over the high-speed backplane to the destination station's card. The backplane typically runs at over 1 Gbps using a proprietary protocol.

What happens if two machines attached to the same plug-in card transmit frames at the same time? It depends on how the card has been constructed. One possibility is for all the ports on the card to be wired together to form a local on-card LAN. Collisions on this on-card LAN will be detected and handled the same as any other collisions on a CSMA/DC network—with retransmissions using the binary backoff algorithm. With this kind of plug-in card, only one transmission per card is possible at any instant, but all the cards can be transmitting in parallel. With this design, each card forms its own **collision domain**, independent of the others.

With the other kind of plug-in card, each input port is buffered, so incoming frames are stored in the card's on-board RAM as they arrive. This design allows all input ports to receive (and transmit) frames at the same time, for parallel, full-duplex operation. Once a frame has been completely received, the card can then check to see if the frame is destined for another port on the same card, or for a distant port. In the former case it can be transmitted directly to the destination. In the latter case, it must be transmitted over the backplane to the proper card. With this design, each port is a separate collision domain, so collisions do not occur. The total system throughput can often be increased by an order of magnitude over 10Base-5, which has a single collision domain for the entire system.

Since the switch just expects standard 802.3 frames on each input port, it is possible use some of the ports as concentrators. In Fig. 4-24, the port in the upper right-hand corner is connected not to a single station, but to a 12-port hub. As frames arrive at the hub, they contend for the 802.3 LAN in the usual way,

including collisions and binary backoff. Successful frames make it to the switch, and are treated there like any other incoming frames: they are switched to the correct output line over the high-speed backplane. If all the input ports are connected to hubs, rather than to individual stations, the switch just becomes an 802.3 to 802.3 bridge. We will study bridges later in this chapter.

4.3.2. IEEE Standard 802.4: Token Bus

Although 802.3 is widely used in offices, during the development of the 802 standard, people from General Motors and other companies interested in factory automation had serious reservations about it. For one thing, due to the probabilistic MAC protocol, with a little bad luck a station might have to wait arbitrarily long to send a frame (i.e., the worst case is unbounded). For another, 802.3 frames do not have priorities, making them unsuited for real-time systems in which important frames should not be held up waiting for unimportant frames.

A simple system with a known worst case is a ring in which the stations take turns sending frames. If there are n stations and it takes T sec to send a frame, no frame will ever have to wait more than nT sec to be sent. The factory automation people in the 802 committee liked the conceptual idea of a ring but did not like the physical implementation because a break in the ring cable would bring the whole network down. Furthermore, they noted that a ring is a poor fit to the linear topology of most assembly lines. As a result, a new standard was developed, having the robustness of the 802.3 broadcast cable, but the known worst-case behavior of a ring.

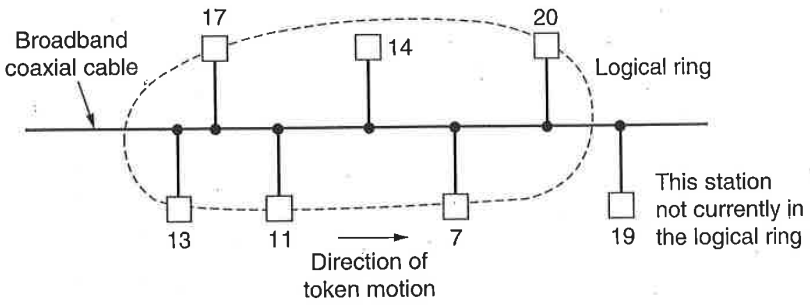


Fig. 4-25. A token bus.

This standard, 802.4 (Dirvin and Miller, 1986; and IEEE, 1985b), describes a LAN called a **token bus**. Physically, the token bus is a linear or tree-shaped cable onto which the stations are attached. Logically, the stations are organized into a ring (see Fig. 4-25), with each station knowing the address of the station to its "left" and "right." When the logical ring is initialized, the highest numbered station may send the first frame. After it is done, it passes permission to its immediate neighbor by sending the neighbor a special control frame called a **token**. The

powered on simultaneously, the protocol deals with this by letting them bid for the token using the standard modified binary countdown algorithm and the two random bits.

Due to transmission errors or hardware failures, problems can arise with the logical ring or the token. For example, if a station tries to pass the token to a station that has gone down, what happens? The solution is straightforward. After passing the token, a station listens to see if its successor either transmits a frame or passes the token. If it does neither, the token is passed a second time.

If that also fails, the station transmits a WHO_FOLLOWS frame specifying the address of its successor. When the failed station's successor sees a WHO_FOLLOWS frame naming its predecessor, it responds by sending a SET_SUCCESSOR frame to the station whose successor failed, naming itself as the new successor. In this way, the failed station is removed from the ring.

Now suppose that a station fails to pass the token to its successor and also fails to locate the successor's successor, which may also be down. It adopts a new strategy by sending a SOLICIT_SUCCESSOR_2 frame to see if *anyone* else is still alive. Once again the standard contention protocol is run, with all stations that want to be in the ring now bidding for a place. Eventually, the ring is reestablished.

Another kind of problem occurs if the token holder goes down and takes the token with it. This problem is solved using the ring initialization algorithm. Each station has a timer that is reset whenever a frame appears on the network. When this timer hits a threshold value, the station issues a CLAIM_TOKEN frame, and the modified binary countdown algorithm with random bits determines who gets the token.

Still another problem is multiple tokens. If a station holding the token notices a transmission from another station, it discards its token. If there were two, there would now be one. If there were more than two, this process would be repeated sooner or later until all but one were discarded. If, by accident, all the tokens are discarded, then the lack of activity will cause one or more stations to try to claim the token.

4.3.3. IEEE Standard 802.5: Token Ring

Ring networks have been around for many years (Pierce, 1972) and have long been used for both local and wide area networks. Among their many attractive features is the fact that a ring is not really a broadcast medium, but a collection of individual point-to-point links that happen to form a circle. Point-to-point links involve a well-understood and field-proven technology and can run on twisted pair, coaxial cable, or fiber optics. Ring engineering is also almost entirely digital, whereas 802.3, for example, has a substantial analog component for collision detection. A ring is also fair and has a known upper bound on channel access.

For these reasons, IBM chose the ring as its LAN and IEEE has included the **token ring** standard as 802.5 (IEEE, 1985c; Latif et al., 1992).

A major issue in the design and analysis of any ring network is the "physical length" of a bit. If the data rate of the ring is R Mbps, a bit is emitted every $1/R$ μsec. With a typical signal propagation speed of about 200 m/μsec, each bit occupies $200/R$ meters on the ring. This means, for example, that a 1-Mbps ring whose circumference is 1000 meters can contain only 5 bits on it at once. The implications of the number of bits on the ring will become clearer later.

As mentioned above, a ring really consists of a collection of ring interfaces connected by point-to-point lines. Each bit arriving at an interface is copied into a 1-bit buffer and then copied out onto the ring again. While in the buffer, the bit can be inspected and possibly modified before being written out. This copying step introduces a 1-bit delay at each interface. A ring and its interfaces are shown in Fig. 4-28.

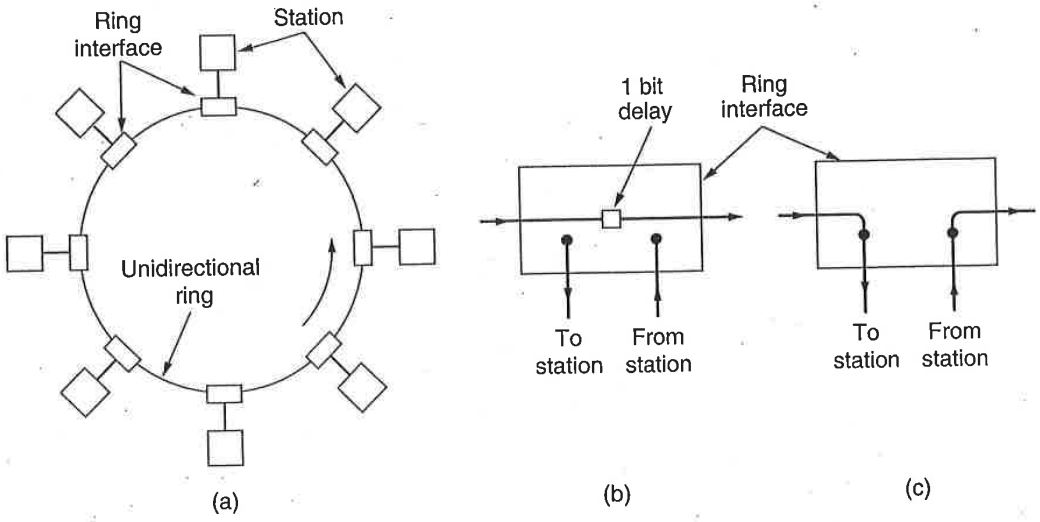


Fig. 4-28. (a) A ring network. (b) Listen mode. (c) Transmit mode.

In a token ring a special bit pattern, called the **token**, circulates around the ring whenever all stations are idle. When a station wants to transmit a frame, it is required to seize the token and remove it from the ring before transmitting. This action is done by inverting a single bit in the 3-byte token, which instantly changes it into the first 3 bytes of a normal data frame. Because there is only one token, only one station can transmit at a given instant, thus solving the channel access problem the same way the token bus solves it.

An implication of the token ring design is that the ring itself must have a sufficient delay to contain a complete token to circulate when all stations are idle. The delay has two components: the 1-bit delay introduced by each station, and the signal propagation delay. In almost all rings, the designers must assume that

stations may be powered down at various times, especially at night. If the interfaces are powered from the ring, shutting down the station has no effect on the interface, but if the interfaces are powered externally, they must be designed to connect the input to the output when power goes down, thus removing the 1-bit delay. The point here is that on a short ring an artificial delay may have to be inserted into the ring at night to ensure that a token can be contained on it.

Ring interfaces have two operating modes, listen and transmit. In listen mode, the input bits are simply copied to output, with a delay of 1 bit time, as shown in Fig. 4-28(b). In transmit mode, which is entered only after the token has been seized, the interface breaks the connection between input and output, entering its own data onto the ring. To be able to switch from listen to transmit mode in 1 bit time, the interface usually needs to buffer one or more frames itself rather than having to fetch them from the station on such short notice.

As bits that have propagated around the ring come back, they are removed from the ring by the sender. The sending station can either save them, to compare with the original data to monitor ring reliability, or discard them. Because the entire frame never appears on the ring at one instant, this ring architecture puts no limit on the size of the frames. After a station has finished transmitting the last bit of its last frame, it must regenerate the token. When the last bit of the frame has gone around and come back, it must be removed, and the interface must switch back into listen mode immediately, to avoid removing the token that might follow if no other station has removed it.

It is straightforward to handle acknowledgements on a token ring. The frame format need only include a 1-bit field for acknowledgements, initially zero. When the destination station has received a frame, it sets the bit. Of course, if the acknowledgement means that the checksum has been verified, the bit must follow the checksum, and the ring interface must be able to verify the checksum as soon as its last bit has arrived. When a frame is broadcast to multiple stations, a more complicated acknowledgement mechanism must be used (if any is used at all).

When traffic is light, the token will spend most of its time idly circulating around the ring. Occasionally a station will seize it, transmit a frame, and then output a new token. However, when the traffic is heavy, so that there is a queue at each station, as soon as a station finishes its transmission and regenerates the token, the next station downstream will see and remove the token. In this manner the permission to send rotates smoothly around the ring, in round-robin fashion. The network efficiency can begin to approach 100 percent under conditions of heavy load.

Now let us turn from token rings in general to the 802.5 standard in particular. At the physical layer, 802.5 calls for shielded twisted pairs running at 1 or 4 Mbps, although IBM later introduced a 16-Mbps version. Signals are encoded using differential Manchester encoding [see Fig. 4-20(c)] with high and low being positive and negative signals of absolute magnitude 3.0 to 4.5 volts. Normally, differential Manchester encoding uses high-low or low-high for each bit, but

802.5 also uses high-high and low-low in certain control bytes (e.g., to mark the start and end of a frame). These nondata signals always occur in consecutive pairs so as not to introduce a DC component into the ring voltage.

One problem with a ring network is that if the cable breaks somewhere, the ring dies. This problem can be solved very elegantly by the use of a **wire center**, as shown in Fig. 4-29. While logically still a ring, physically each station is connected to the wire center by a cable containing (at least) two twisted pairs, one for data to the station and one for data from the station.

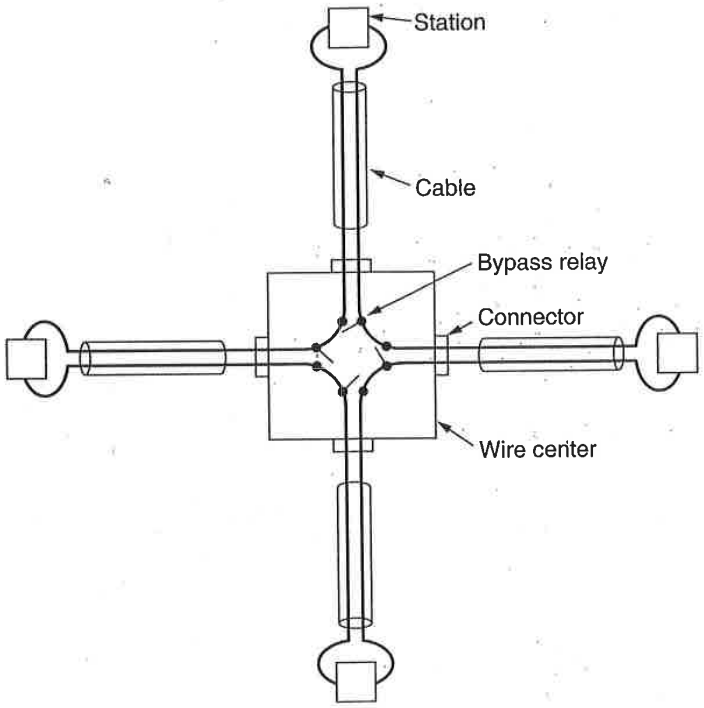


Fig. 4-29. Four stations connected via a wire center.

Inside the wire center are bypass relays that are energized by current from the stations. If the ring breaks or a station goes down, loss of the drive current will release the relay and bypass the station. The relays can also be operated by software to permit diagnostic programs to remove stations one at a time to find faulty stations and ring segments. The ring can then continue operation with the bad segment bypassed. Although the 802.5 standard does not formally require this kind of ring, often called a **star-shaped ring** (Saltzer et al., 1983), most 802.5 LANs, in fact, do use wire centers to improve their reliability and maintainability.

When a network consists of many clusters of stations far apart, a topology with multiple wire centers can be used. Just imagine that the cable to one of the stations in Fig. 4-29 were replaced by a cable to a distant wire center. Although logically all the stations are on the same ring, the wiring requirements are greatly

reduced. An 802.5 ring using a wire center has a similar topology to an 802.3 10Base-T hub-based network, but the formats and protocols are different.

The Token Ring MAC Sublayer Protocol

The basic operation of the MAC protocol is straightforward. When there is no traffic on the ring, a 3-byte token circulates endlessly, waiting for a station to seize it by setting a specific 0 bit to a 1 bit, thus converting the token into the start-of-frame sequence. The station then outputs the rest of a normal data frame, as shown in Fig. 4-30.

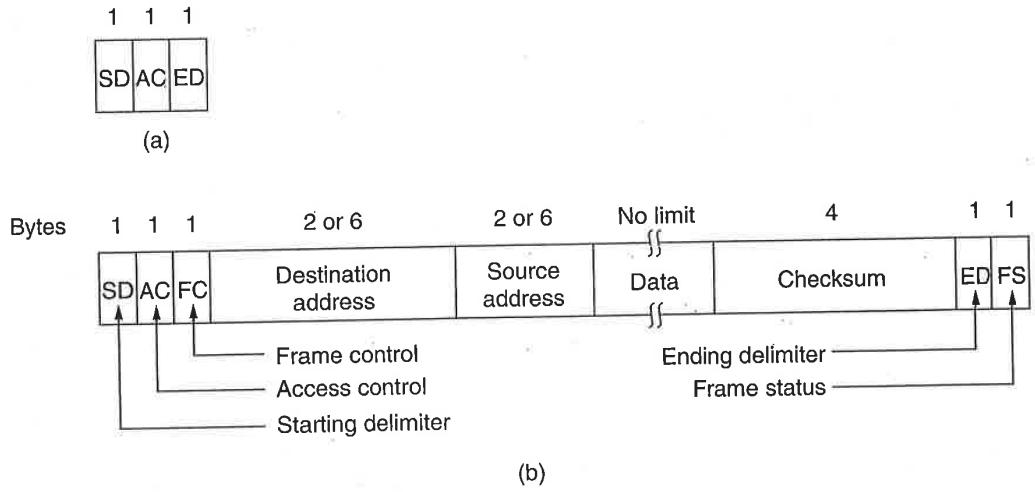


Fig. 4-30. (a) Token format. (b) Data frame format.

Under normal conditions, the first bit of the frame will go around the ring and return to the sender before the full frame has been transmitted. Only a very long ring will be able to hold even a short frame. Consequently, the transmitting station must drain the ring while it continues to transmit. As shown in Fig. 4-28(c), this means that the bits that have completed the trip around the ring come back to the sender and are removed.

A station may hold the token for the **token-holding time**, which is 10 msec unless an installation sets a different value. If there is enough time left after the first frame has been transmitted to send more frames, these may be sent as well. After all pending frames have been transmitted or the transmission of another frame would exceed the token-holding time, the station regenerates the 3-byte token frame and puts it out onto the ring.

The *Starting delimiter* and *Ending delimiter* fields of Fig. 4-30(b) mark the beginning and ending of the frame. Each contains invalid differential Manchester patterns (HH and LL) to distinguish them from data bytes. The *Access control*

byte contains the token bit, and also the *Monitor bit*, *Priority bits*, and *Reservation bits* (described below). The *Frame control* byte distinguishes data frames from various possible control frames.

Next come the *Destination address* and *Source address* fields, which are the same as in 802.3 and 802.4. These are followed by the data, which may be as long as necessary, provided that the frame can still be transmitted within the token-holding time. The *Checksum* field, like the destination and source addresses, is also the same as 802.3 and 802.4.

An interesting byte not present in the other two protocols is the *Frame status* byte. It contains the *A* and *C* bits. When a frame arrives at the interface of a station with the destination address, the interface turns on the *A* bit as it passes through. If the interface copies the frame to the station, it also turns on the *C* bit. A station might fail to copy a frame due to lack of buffer space or other reasons.

When the sending station drains the frame from the ring, it examines the *A* and *C* bits. Three combinations are possible:

1. $A = 0$ and $C = 0$: destination not present or not powered up.
2. $A = 1$ and $C = 0$: destination present but frame not accepted.
3. $A = 1$ and $C = 1$: destination present and frame copied.

This arrangement provides an automatic acknowledgement for each frame. If a frame is rejected but the station is present, the sender has the option of trying again in a little while. The *A* and *C* bits are present twice in the *Frame status* to increase reliability inasmuch as they are not covered by the checksum.

The *Ending delimiter* contains an *E* bit which is set if any interface detects an error (e.g., a non-Manchester pattern where that is not permitted). It also contains a bit that can be used to mark the last frame in a logical sequence, sort of like an end-of-file bit.

The 802.5 protocol has an elaborate scheme for handling multiple priority frames. The 3-byte token frame contains a field in the middle byte giving the priority of the token. When a station wants to transmit a priority n frame, it must wait until it can capture a token whose priority is less than or equal to n. Furthermore, when a data frame goes by, a station can try to reserve the next token by writing the priority of the frame it wants to send into the frame's *Reservation bits*. However, if a higher priority has already been reserved there, the station may not make a reservation. When the current frame is finished, the next token is generated at the priority that has been reserved.

A little thought will show that this mechanism acts like a ratchet, always jacking the reservation priority higher and higher. To eliminate this problem, the protocol contains some complex rules. The essence of the idea is that the station raising the priority is responsible for lowering the priority again when it is done.

Notice that this priority scheme is substantially different from the token bus scheme, in which each station always gets its fair share of the bandwidth, no

matter what other stations are doing. In the token ring, a station with only low priority frames may starve to death waiting for a low priority token to appear. Clearly, the two committees had different taste when trading off good service for high priority traffic versus fairness to all stations.

Ring Maintenance

The token bus protocol goes to considerable lengths to do ring maintenance in a fully decentralized way. The token ring protocol handles maintenance quite differently. Each token ring has a **monitor station** that oversees the ring. If the monitor goes down, a contention protocol ensures that another station is quickly elected as monitor. (Every station has the capability of becoming the monitor.) While the monitor is functioning properly, it alone is responsible for seeing that the ring operates correctly.

When the ring comes up or any station notices that there is no monitor, it can transmit a CLAIM TOKEN control frame. If this frame circumnavigates the ring before any other CLAIM TOKEN frames are sent, the sender becomes the new monitor (each station has monitor capability built in). The token ring control frames are shown in Fig. 4-31.

Control field	Name	Meaning
00000000	Duplicate address test	Test if two stations have the same address
00000010	Beacon	Used to locate breaks in the ring
00000011	Claim token	Attempt to become monitor
00000100	Purge	Reinitialize the ring
00000101	Active monitor present	Issued periodically by the monitor
00000110	Standby monitor present	Announces the presence of potential monitors

Fig. 4-31. Token ring control frames.

Among the monitor's responsibilities are seeing that the token is not lost, taking action when the ring breaks, cleaning the ring up when garbled frames appear, and watching out for orphan frames. An orphan frame occurs when a station transmits a short frame in its entirety onto a long ring and then crashes or is powered down before the frame can be drained. If nothing is done, the frame will circulate forever.

To check for lost tokens, the monitor has a timer that is set to the longest possible tokenless interval: each station transmitting for the full token-holding time. If this timer goes off, the monitor drains the ring and issues a new token.

When a garbled frame appears, the monitor can detect it by its invalid format or checksum, open the ring to drain it, and issue a new token when the ring has

been cleaned up. Finally, the monitor detects orphan frames by setting the *monitor* bit in the *Access control* byte whenever it passes through. If an incoming frame has this bit set, something is wrong since the same frame has passed the monitor twice without having been drained, so the monitor drains it.

One last monitor function concerns the length of the ring. The token is 24 bits long, which means that the ring must be big enough to hold 24 bits. If the 1-bit delays in the stations plus the cable length add up to less than 24 bits, the monitor inserts extra delay bits so that a token can circulate.

One maintenance function that cannot be handled by the monitor is locating breaks in the ring. When a station notices that either of its neighbors appears to be dead, it transmits a BEACON frame giving the address of the presumably dead station. When the beacon has propagated around as far as it can, it is then possible to see how many stations are down and delete them from the ring using the bypass relays in the wire center, all without human intervention.

It is instructive to compare the approaches taken to controlling the token bus and the token ring. The 802.4 committee was scared to death of having any centralized component that could fail in some unexpected way and take the system down with it. Therefore they designed a system in which the current token holder had special powers (e.g., soliciting bids to join the ring), but no station was otherwise different from the others (e.g., currently assigned administrative responsibility for maintenance).

The 802.5 committee, on the other hand, felt that having a centralized monitor made handling lost tokens, orphan frames and so on much easier. Furthermore, in a normal system, stations hardly ever crash, so occasionally having to put up with contention for a new monitor is not a great hardship. The price paid is that if the monitor ever really goes berserk but continues to issue ACTIVE MONITOR PRESENT control frames periodically, no station will ever challenge it. Monitors cannot be impeached.

This difference in approach comes from the different application areas the two committees had in mind. The 802.4 committee was thinking in terms of factories with large masses of metal moving around under computer control. Network failures could result in severe damage and had to be prevented at all costs. The 802.5 committee was interested in office automation, where a failure once in a rare while could be tolerated as the price for a simpler system. Whether 802.4 is, in fact, more reliable than 802.5 is a matter of some controversy.

4.3.4. Comparison of 802.3, 802.4, and 802.5

With three different and incompatible LANs available, each with different properties, many organizations are faced with the question: Which one should we install? In this section we will look at all three of the 802 LAN standards, pointing out their strengths and weaknesses, comparing and contrasting them.

4.5.1. FDDI

FDDI (Fiber Distributed Data Interface) is a high-performance fiber optic token ring LAN running at 100 Mbps over distances up to 200 km with up to 1000 stations connected (Black, 1994; Jain, 1994; Mirchandani and Khanna, 1993; Ross and Hamstra, 1993; Shah and Ramakrishnan, 1994; and Wolter, 1990). It can be used in the same way as any of the 802 LANs, but with its high bandwidth, another common use is as a backbone to connect copper LANs, as shown in Fig. 4-44. FDDI-II is the successor to FDDI, modified to handle synchronous circuit-switched PCM data for voice or ISDN traffic, in addition to ordinary data. We will refer to both of them as just FDDI. This section deals with both the physical layer and the MAC sublayer of FDDI.

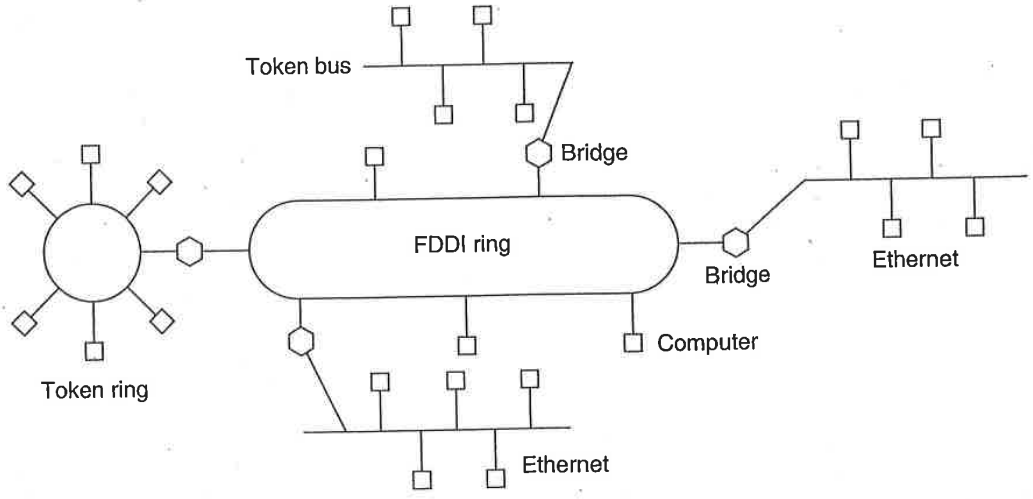


Fig. 4-44. An FDDI ring being used as a backbone to connect LANs and computers.

FDDI uses multimode fibers because the additional expense of single mode fibers is not needed for networks running at only 100 Mbps. It also uses LEDs rather than lasers, not only due to their lower cost, but also because FDDI may sometimes be used to connect directly to user workstations. There is a danger that curious users may occasionally unplug the fiber connector and look directly into it to watch the bits go by at 100 Mbps. With a laser the curious user might end up with a hole in his retina. LEDs are too weak to do any eye damage but are strong enough to transfer data accurately at 100 Mbps. The FDDI design specification calls for no more than 1 error in 2.5×10^{10} bits. Many implementations do much better.

The FDDI cabling consists of two fiber rings, one transmitting clockwise and the other transmitting counterclockwise, as illustrated in Fig. 4-45(a). If either one breaks, the other can be used as a backup. If both break at the same point, for

example, due to a fire or other accident in the cable duct, the two rings can be joined into a single ring approximately twice as long, as shown in Fig. 4-45(b). Each station contains relays that can be used to join the two rings or bypass the station in the event of station problems. Wire centers can also be used, as in 802.5.

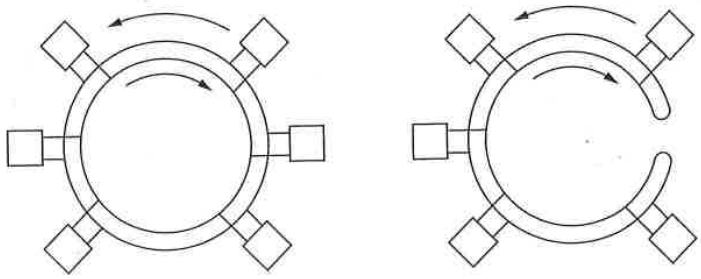


Fig. 4-45. (a) FDDI consists of two counterrotating rings. (b) In the event of failure of both rings at one point, the two rings can be joined together to form a single long ring.

FDDI defines two classes of stations, *A* and *B*. Class *A* stations connect to both rings. The cheaper class *B* stations only connect to one of the rings. Depending on how important fault tolerance is, an installation can choose class *A* or class *B* stations, or some of each.

The physical layer does not use Manchester encoding because 100-Mbps Manchester encoding requires 200 megabaud, which was deemed too expensive. Instead a scheme called **4 out of 5** encoding is used. Each group of 4 MAC symbols (0s, 1s, and certain nondata symbols such as start-of-frame) are encoded as a group of 5 bits on the medium. Sixteen of the 32 combinations are for data, 3 are for delimiters, 2 are for control, 3 are for hardware signaling, and 8 are unused (i.e., reserved for future versions of the protocol).

The advantage of this scheme is that it saves bandwidth, but the disadvantage is the loss of the self-clocking property of Manchester encoding. To compensate for this loss, a long preamble is used to synchronize the receiver to the sender's clock. Furthermore, all clocks are required to be stable to at least 0.005 percent. With this stability, frames up to 4500 bytes can be sent without danger of the receiver's clock drifting too far out of sync with the data stream.

The basic FDDI protocols are closely modeled on the 802.5 protocols. To transmit data, a station must first capture the token. Then it transmits a frame and removes it when it comes around again. One difference between FDDI and 802.5 is that in 802.5, a station may not generate a new token until its frame has gone all the way around and come back. In FDDI, with potentially 1000 stations and 200 km of fiber, the amount of time wasted waiting for the frame to circumnavigate the ring could be substantial. For this reason, it was decided to allow a station to

put a new token back onto the ring as soon as it has finished transmitting its frames. In a large ring, several frames might be on the ring at the same time.

FDDI data frames are similar to 802.5 data frames. The FDDI format is shown in Fig. 4-46. The *Start delimiter* and *End delimiter* fields mark the frame boundaries. The *Frame control* field tells what kind of frame this is (data, control, etc.). The *Frame status* byte holds acknowledgement bits, similar to those of 802.5. The other fields are analogous to 802.5.

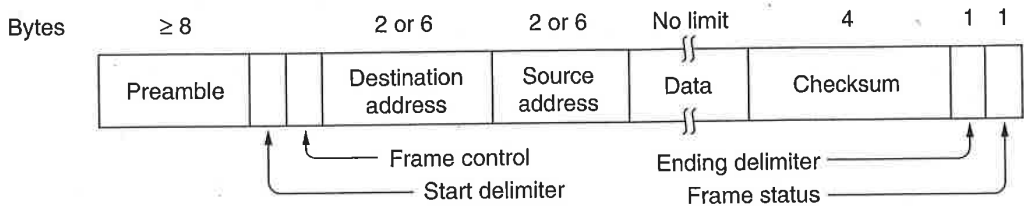


Fig. 4-46. FDDI frame format.

In addition to the regular (asynchronous) frames, FDDI also permits special synchronous frames for circuit-switched PCM or ISDN data. The synchronous frames are generated every 125 μsec by a master station to provide the 8000 samples/sec needed by PCM systems. Each of these frames has a header, 16 bytes of noncircuit-switched data, and up to 96 bytes of circuit-switched data (i.e., up to 96 PCM channels per frame).

The number 96 was chosen because it allows four T1 channels (4×24) at 1.544 Mbps or three CCITT E1 channels (3×32) at 2.048 Mbps to fit in a frame, thus making it suitable for use anywhere in the world. One synchronous frame every 125 μsec consumes 6.144 Mbps of bandwidth for the 96 circuit-switched channels. A maximum of 16 synchronous frames every 125 μsec allows up to 1536 PCM channels and eats up 98.3 Mbps.

Once a station has acquired one or more time slots in a synchronous frame, those slots are reserved for it until they are explicitly released. The total bandwidth not used by the synchronous frames is allocated on demand. A bit mask is present in each of these frames to indicate which slots are available for demand assignment. The nonsynchronous traffic is divided into priority classes, with the higher priorities getting first shot at the leftover bandwidth.

The FDDI MAC protocol uses three timers. The **token holding timer** determines how long a station may continue to transmit once it has acquired the token. This timer prevents a station from hogging the ring forever. The **token rotation timer** is restarted every time the token is seen. If this timer expires, it means that the token has not been sighted for too long an interval. Probably it has been lost, so the token recovery procedure is initiated. Finally, the **valid transmission timer** is used to time out and recover from certain transient ring errors.

FDDI also has a priority algorithm similar to 802.4. It determines which

priority classes may transmit on a given token pass. If the token is ahead of schedule, all priorities may transmit, but if it is behind schedule, only the highest ones may send.

4.5.2. Fast Ethernet

FDDI was supposed to be the next generation LAN, but it never really caught on much beyond the backbone market (where it continues to do fine). The station management was too complicated, which led to complex chips and high prices. The substantial cost of FDDI chips made workstation manufacturers unwilling to make FDDI the standard network, so volume production never happened and FDDI never broke through to the mass market. The lesson that should have been learned here was KISS (Keep It Simple, Stupid).

In any event, the failure of FDDI to catch fire left a gap for a garden-variety LAN at speeds above 10 Mbps. Many installations needed more bandwidth and thus had numerous 10-Mbps LANs connected by a maze of repeaters, bridges, routers, and gateways, although to the network managers it sometimes felt that they were being held together by bubble gum and chicken wire.

It was in this environment that IEEE reconvened the 802.3 committee in 1992 with instructions to come up with a faster LAN. One proposal was to keep 802.3 exactly as it was, but just make it go faster. Another proposal was to redo it totally, to give it lots of new features, such as real-time traffic and digitized voice, but just keep the old name (for marketing reasons). After some wrangling, the committee decided to keep 802.3 the way it was, but just make it go faster. The people behind the losing proposal did what any computer-industry people would have done under these circumstances—they formed their own committee and standardized their LAN anyway (eventually as 802.12).

The three primary reasons that the 802.3 committee decided to go with a souped-up 802.3 LAN were:

1. The need to be backward compatible with thousands of existing LANs.
2. The fear that a new protocol might have unforeseen problems.
3. The desire to get the job done before the technology changed.

The work was done quickly (by standards committees' norms), and the result, **802.3u**, was officially approved by IEEE in June 1995. Technically, 802.3u is not a new standard, but an addendum to the existing 802.3 standard (to emphasize its backward compatibility). Since everyone calls it **fast Ethernet**, rather than 802.3u, we will do that, too.

The basic idea behind fast Ethernet was simple: keep all the old packet formats, interfaces, and procedural rules, but just reduce the bit time from 100 nsec to 10 nsec. Technically, it would have been possible to copy 10Base-5 or 10Base-2 and still detect collisions on time by just reducing the maximum cable

the actions following the wakeup also count. For example, if a CALL REQUEST packet comes in and a process was asleep waiting for it, the transmission of the CALL ACCEPT packet following the wakeup counts as part of the action for CALL REQUEST. After each action is performed, the connection may move to a new state, as shown in Fig. 6-21.

The advantage of representing the protocol as a matrix is threefold. First, in this form it is much easier for the programmer to systematically check each combination of state and event to see if an action is required. In production implementations, some of the combinations would be used for error handling. In Fig. 6-21 no distinction is made between impossible situations and illegal ones. For example, if a connection is in *waiting* state, the DISCONNECT event is impossible because the user is blocked and cannot execute any primitives at all. On the other hand, in *sending* state, data packets are not expected because no credit has been issued. The arrival of a data packet is a protocol error.

The second advantage of the matrix representation of the protocol is in implementing it. One could envision a two-dimensional array in which element $a[i][j]$ was a pointer or index to the procedure that handled the occurrence of event i when in state j. One possible implementation is to write the transport entity as a short loop, waiting for an event at the top of the loop. When an event happens, the relevant connection is located and its state is extracted. With the event and state now known, the transport entity just indexes into the array a and calls the proper procedure. This approach gives a much more regular and systematic design than our transport entity.

The third advantage of the finite state machine approach is for protocol description. In some standards documents, the protocols are given as finite state machines of the type of Fig. 6-21. Going from this kind of description to a working transport entity is much easier if the transport entity is also driven by a finite state machine based on the one in the standard.

The primary disadvantage of the finite state machine approach is that it may be more difficult to understand than the straight programming example we used initially. However, this problem may be partially solved by drawing the finite state machine as a graph, as is done in Fig. 6-22.

6.4. THE INTERNET TRANSPORT PROTOCOLS (TCP AND UDP)

The Internet has two main protocols in the transport layer, a connection-oriented protocol and a connectionless one. In the following sections we will study both of them. The connection-oriented protocol is TCP. The connectionless protocol is UDP. Because UDP is basically just IP with a short header added, we will focus on TCP.

TCP (Transmission Control Protocol) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An

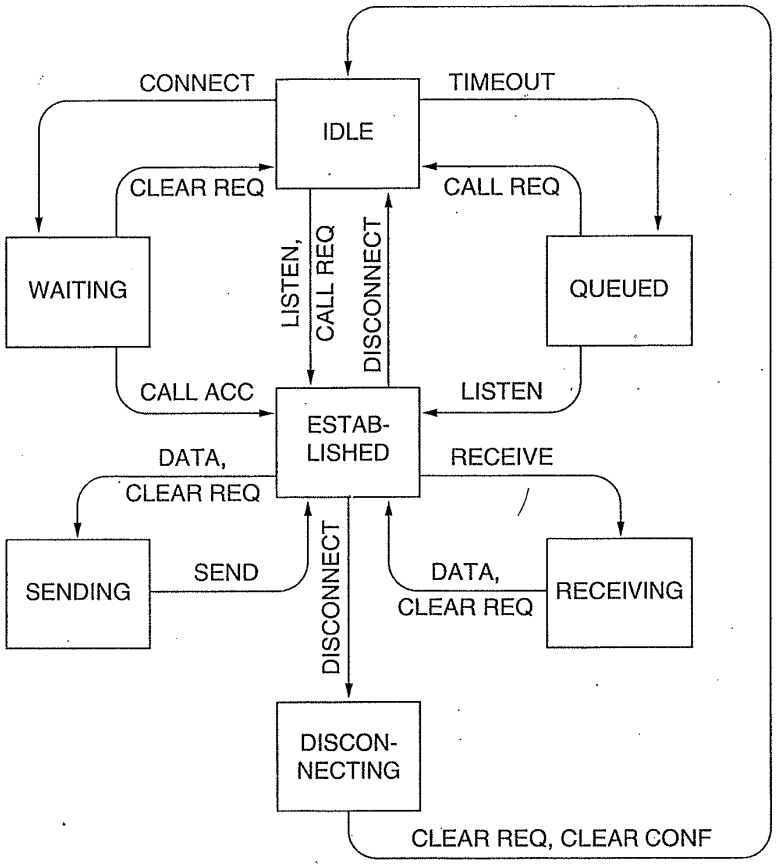


Fig. 6-22. The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

TCP was formally defined in RFC 793. As time went on, various errors and inconsistencies were detected, and the requirements were changed in some areas. These clarifications and some bug fixes are detailed in RFC 1122. Extensions are given in RFC 1323.

Each machine supporting TCP has a TCP transport entity, either a user process or part of the kernel that manages TCP streams and interfaces to the IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64K bytes (in practice, usually about 1500 bytes), and sends each piece as a separate IP datagram. When IP datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams. For simplicity, we will sometimes use just "TCP" to mean the

TCP transport entity (a piece of software) or the TCP protocol (a set of rules). From the context it will be clear which is meant. For example, in "The user gives TCP the data," the TCP transport entity is clearly intended.

The IP layer gives no guarantee that datagrams will be delivered properly, so it is up to TCP to time out and retransmit them as need be. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence. In short, TCP must furnish the reliability that most users want and that IP does not provide.

6.4.1. The TCP Service Model

TCP service is obtained by having both the sender and receiver create end points, called sockets, as discussed in Sec. 6.1.3. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a **port**. A port is the TCP name for a TSAP. To obtain TCP service, a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine. The socket calls are listed in Fig. 6-6.

A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket. Connections are identified by the socket identifiers at both ends, that is, (*socket1*, *socket2*). No virtual circuit numbers or other identifiers are used.

Port numbers below 256 are called **well-known ports** and are reserved for standard services. For example, any process wishing to establish a connection to a host to transfer a file using FTP can connect to the destination host's port 21 to contact its FTP daemon. Similarly, to establish a remote login session using TELNET, port 23 is used. The list of well-known ports is given in RFC 1700.

All TCP connections are full-duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end. For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk (see Fig. 6-23), or some other way. There is no way for the receiver to detect the unit(s) in which the data were written.

Files in UNIX have this property too. The reader of a file cannot tell whether the file was written a block at a time, a byte at a time, or all in one blow. As with a UNIX file, the TCP software has no idea of what the bytes mean and no interest in finding out. A byte is just a byte.

When an application passes data to TCP, TCP may send it immediately or buffer it (in order to collect a larger amount to send at once), at its discretion.

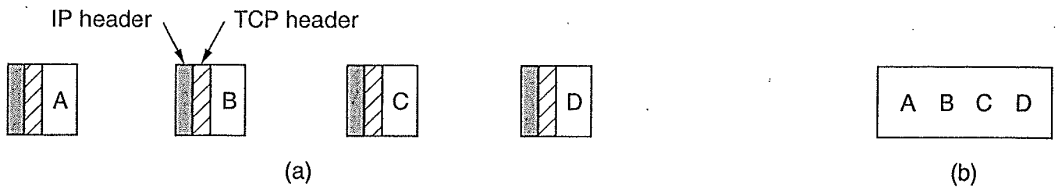


Fig. 6-23. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

However, sometimes, the application really wants the data to be sent immediately. For example, suppose a user is logged into a remote machine. After a command line has been finished and the carriage return typed, it is essential that the line be shipped off to the remote machine immediately and not buffered until the next line comes in. To force data out, applications can use the PUSH flag, which tells TCP not to delay the transmission.

Some early applications used the PUSH flag as a kind of marker to delineate messages boundaries. While this trick sometimes works, it sometimes fails since not all implementations of TCP pass the PUSH flag to the application on the receiving side. Furthermore, if additional PUSHes come in before the first one has been transmitted (e.g., because the output line is busy), TCP is free to collect all the PUSHed data into a single IP datagram, with no separation between the various pieces.

One last feature of the TCP service that is worth mentioning here is **urgent data**. When an interactive user hits the DEL or CTRL-C key to break off a remote computation that has already begun, the sending application puts some control information in the data stream and gives it to TCP along with the URGENT flag. This event causes TCP to stop accumulating data and transmit everything it has for that connection immediately.

When the urgent data are received at the destination, the receiving application is interrupted (e.g., given a signal in UNIX terms), so it can stop whatever it was doing and read the data stream to find the urgent data. The end of the urgent data is marked, so the application knows when it is over. The start of the urgent data is not marked. It is up to the application to figure that out. This scheme basically provides a crude signaling mechanism and leaves everything else up to the application.

6.4.2. The TCP Protocol

In this section we will give a general overview of the TCP protocol. In the next one we will go over the protocol header, field by field. Every byte on a TCP connection has its own 32-bit sequence number. For a host blasting away at full

speed on a 10-Mbps LAN, theoretically the sequence numbers could wrap around in an hour, but in practice it takes much longer. The sequence numbers are used both for acknowledgements and for the window mechanism, which use separate 32-bit header fields.

The sending and receiving TCP entities exchange data in the form of segments. A **segment** consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,535 byte IP payload. Second, each network has a **maximum transfer unit** or **MTU**, and each segment must fit in the MTU. In practice, the MTU is generally a few thousand bytes and thus defines the upper bound on segment size. If a segment passes through a sequence of networks without being fragmented and then hits one whose MTU is smaller than the segment, the router at the boundary fragments the segment into two or more smaller segments.

A segment that is too large for a network that it must transit can be broken up into multiple segments by a router. Each new segment gets its own TCP and IP headers, so fragmentation by routers increases the total overhead (because each additional segment adds 40 bytes of extra header information).

The basic protocol used by TCP entities is the sliding window protocol. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exists, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

Although this protocol sounds simple, there are many ins and outs that we will cover below. For example, since segments can be fragmented, it is possible that part of a transmitted segment arrives and is acknowledged by the receiving TCP entity, but the rest is lost. Segments can also arrive out of order, so bytes 3072–4095 can arrive but cannot be acknowledged because bytes 2048–3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. If a retransmitted segment takes a different route than the original, and is fragmented differently, bits and pieces of both the original and the duplicate can arrive sporadically, requiring a careful administration to achieve a reliable byte stream. Finally, with so many networks making up the Internet, it is possible that a segment may occasionally hit a congested (or broken) network along its path.

TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems. A number of the algorithms used by many TCP implementations will be discussed below.

6.4.3. The TCP Segment Header

Figure 6-24 shows the layout of a TCP segment. Every segment begins with a fixed-format 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 – 20 – 20 = 65,515 data bytes may follow, where the first 20 refers to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.

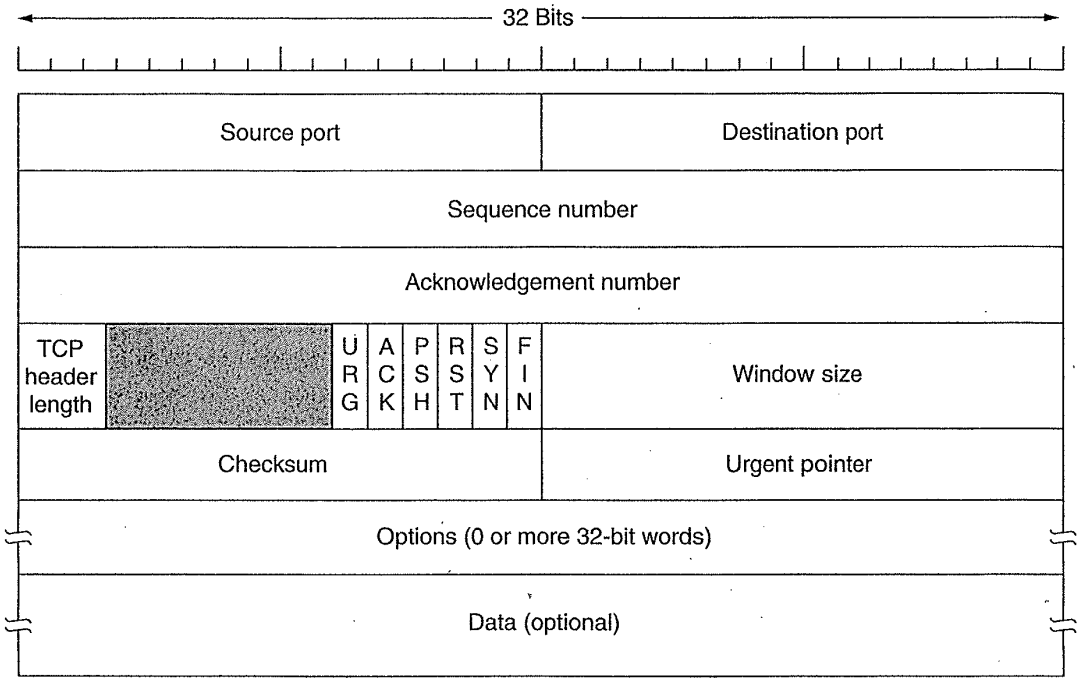


Fig. 6-24. The TCP header.

Let us dissect the TCP header field by field. The *Source port* and *Destination port* fields identify the local end points of the connection. Each host may decide for itself how to allocate its own ports starting at 256. A port plus its host's IP address forms a 48-bit unique TSAP. The source and destination socket numbers together identify the connection.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions. Note that the latter specifies the next byte expected, not the last byte correctly received. Both are 32 bits long because every byte of data is numbered in a TCP stream.

The *TCP header length* tells how many 32-bit words are contained in the TCP header. This information is needed because the *Options* field is of variable length, so the header is too. Technically, this field really indicates the start of the data

within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same.

Next comes a 6-bit field that is not used. The fact that this field has survived intact for over a decade is testimony to how well thought out TCP is. Lesser protocols would have needed it to fix bugs in the original design.

Now come six 1-bit flags. *URG* is set to 1 if the *Urgent pointer* is in use. The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found. This facility is in lieu of interrupt messages. As we mentioned above, this facility is a bare bones way of allowing the sender to signal the receiver without getting TCP itself involved in the reason for the interrupt.

The *ACK* bit is set to 1 to indicate that the *Acknowledgement number* is valid. If *ACK* is 0, the segment does not contain an acknowledgement so the *Acknowledgement number* field is ignored.

The *PSH* bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency reasons).

The *RST* bit is used to reset a connection that has become confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection. In general, if you get a segment with the *RST* bit on, you have a problem on your hands.

The *SYN* bit is used to establish connections. The connection request has $SYN = 1$ and $ACK = 0$ to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, so it has $SYN = 1$ and $ACK = 1$. In essence the *SYN* bit is used to denote CONNECTION REQUEST and CONNECTION ACCEPTED, with the *ACK* bit used to distinguish between those two possibilities.

The *FIN* bit is used to release a connection. It specifies that the sender has no more data to transmit. However, after closing a connection, a process may continue to receive data indefinitely. Both *SYN* and *FIN* segments have sequence numbers and are thus guaranteed to be processed in the correct order.

Flow control in TCP is handled using a variable-size sliding window. The *Window* field tells how many bytes may be sent starting at the byte acknowledged. A *Window* field of 0 is legal and says that the bytes up to and including *Acknowledgement number* − 1 have been received, but that the receiver is currently badly in need of a rest and would like no more data for the moment, thank you. Permission to send can be granted later by sending a segment with the same *Acknowledgement number* and a nonzero *Window* field.

A *Checksum* is also provided for extreme reliability. It checksums the header, the data, and the conceptual pseudoheader shown in Fig. 6-25. When performing this computation, the TCP *Checksum* field is set to zero, and the data field is padded out with an additional zero byte if its length is an odd number. The checksum algorithm is simply to add up all the 16-bit words in 1's complement and then to

take the 1's complement of the sum. As a consequence, when the receiver performs the calculation on the entire segment, including the *Checksum* field, the result should be 0.

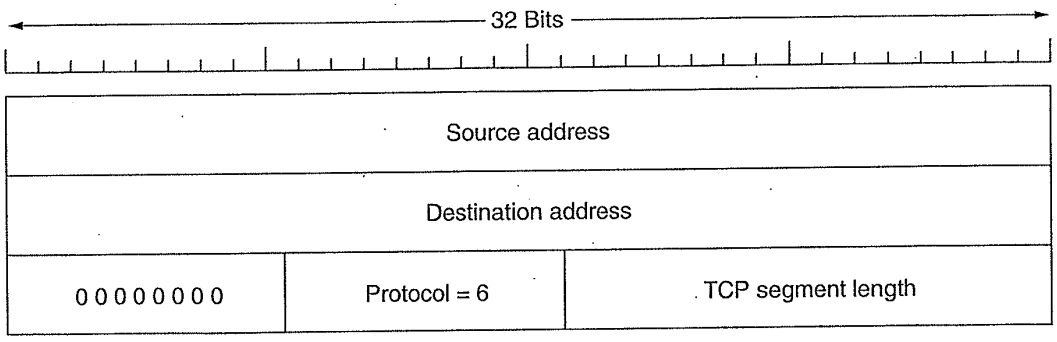


Fig. 6-25. The pseudoheader included in the TCP checksum.

The pseudoheader contains the 32-bit IP addresses of the source and destination machines, the protocol number for TCP (6), and the byte count for the TCP segment (including the header). Including the pseudoheader in the TCP checksum computation helps detect misdelivered packets, but doing so violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not the TCP layer.

The *Options* field was designed to provide a way to add extra facilities not covered by the regular header. The most important option is the one that allows each host to specify the maximum TCP payload it is willing to accept. Using large segments is more efficient than using small ones because the 20-byte header can then be amortized over more data, but small hosts may not be able to handle very large segments. During connection setup, each side can announce its maximum and see its partner's. The smaller of the two numbers wins. If a host does not use this option, it defaults to a 536-byte payload. All Internet hosts are required to accept TCP segments of 536 + 20 = 556 bytes.

For lines with high bandwidth, high delay, or both, the 64 KB window is often a problem. On a T3 line (44.736 Mbps), it takes only 12 msec to output a full 64 KB window. If the round trip propagation delay is 50 msec (typical for a transcontinental fiber), the sender will be idle 3/4 of the time waiting for acknowledgements. On a satellite connection, the situation is even worse. A larger window size would allow the sender to keep pumping data out, but using the 16-bit *Window size* field, there is no way to express such a size. In RFC 1323, a *Window scale* option was proposed, allowing the sender and receiver to negotiate a window scale factor. This number allows both sides to shift the *Window size* field up to 16 bits to the left, thus allowing windows of up to 2^{32} bytes. Most TCP implementations now support this option.

Another option proposed by RFC 1106 and now widely implemented is the use of the selective repeat instead of go back n protocol. If the receiver gets one bad segment and then a large number of good ones, the normal TCP protocol will

eventually time out and retransmit all the unacknowledged segments, including all those that were received correctly. RFC 1106 introduced NAKs, to allow the receiver to ask for a specific segment (or segments). After it gets these, it can acknowledge all the buffered data, thus reducing the amount of data retransmitted.

6.4.4. TCP Connection Management

Connections are established in TCP using the three-way handshake discussed in Sec. 6.2.2. To establish a connection, one side, say the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives, either specifying a specific source or nobody in particular.

The other side, say the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the *SYN* bit on and *ACK* bit off and waits for a response.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the *Destination port* field. If not, it sends a reply with the *RST* bit on to reject the connection.

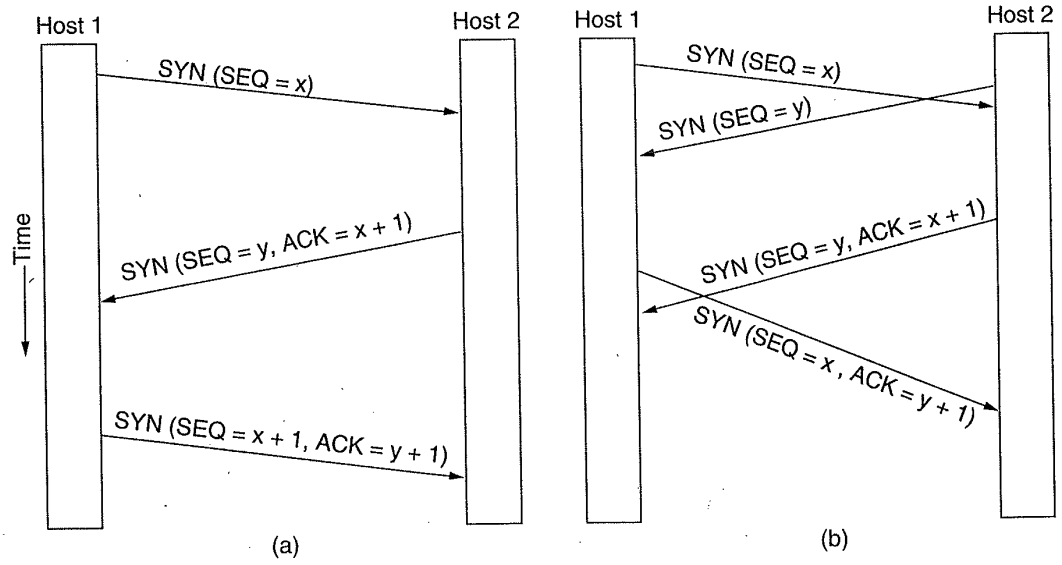


Fig. 6-26. (a) TCP connection establishment in the normal case. (b) Call collision.

If some process is listening to the port, that process is given the incoming TCP segment. It can then either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig. 6-26(a). Note that a *SYN* segment consumes 1 byte of sequence space so it can be acknowledged unambiguously.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig. 6-26(b). The result of these events is that just one connection is established, not two because connections are identified by their end points. If the first setup results in a connection identified by (x, y) and the second one does too, only one table entry is made, namely, for (x, y).

The initial sequence number on a connection is not 0 for the reasons we discussed earlier. A clock-based scheme is used, with a clock tick every 4 μsec. For additional safety, when a host crashes, it may not reboot for the maximum packet lifetime (120 sec) to make sure that no packets from previous connections are still roaming around the Internet somewhere.

Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling. To release a connection, either party can send a TCP segment with the *FIN* bit set, which means that it has no more data to transmit. When the *FIN* is acknowledged, that direction is shut down. Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection, one *FIN* and one *ACK* for each direction. However, it is possible for the first *ACK* and the second *FIN* to be contained in the same segment, reducing the total count to three.

Just as with telephone calls in which both people say goodbye and hang up the phone simultaneously, both ends of a TCP connection may send *FIN* segments at the same time. These are each acknowledged in the usual way, and the connection shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously.

To avoid the two-army problem, timers are used. If a response to a *FIN* is not forthcoming within two maximum packet lifetimes, the sender of the *FIN* releases the connection. The other side will eventually notice that nobody seems to be listening to it any more, and time out as well. While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

The steps required to establish and release connections can be represented in a finite state machine with the 11 states listed in Fig. 6-27. In each state, certain events are legal. When a legal event happens, some action may be taken. If some other event happens, an error is reported.

Each connection starts in the *CLOSED* state. It leaves that state when it does either a passive open (LISTEN), or an active open (CONNECT). If the other side does the opposite one, a connection is established and the state becomes *ESTABLISHED*. Connection release can be initiated by either side. When it is complete, the state returns to *CLOSED*.

The finite state machine itself is shown in Fig. 6-28. The common case of a client actively connecting to a passive server is shown with heavy lines—solid for

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Fig. 6-27. The states used in the TCP connection management finite state machine.

the client, dotted for the server. The lightface lines are unusual event sequences. Each line in Fig. 6-28 is marked by an *event/action* pair. The event can either be a user-initiated system call (CONNECT, LISTEN, SEND, or CLOSE), a segment arrival (*SYN*, *FIN*, *ACK*, or *RST*), or in one case, a timeout of twice the maximum packet lifetime. The action is the sending of a control segment (*SYN*, *FIN*, or *RST*) or nothing, indicated by —. Comments are shown in parentheses.

The diagram can best be understood by first following the path of a client (the heavy solid line) then later the path of a server (the heavy dashed line). When an application on the client machine issues a CONNECT request, the local TCP entity creates a connection record, marks it as being in the *SYN SENT* state, and sends a *SYN* segment. Note that many connections may be open (or being opened) at the same time on behalf of multiple applications, so the state is per connection and recorded in the connection record. When the *SYN+ACK* arrives, TCP sends the final *ACK* of the three-way handshake and switches into the *ESTABLISHED* state. Data can now be sent and received.

When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a *FIN* segment and wait for the corresponding *ACK* (dashed box marked active close). When the *ACK* arrives, a transition is made to state *FIN WAIT 2* and one direction of the connection is now closed. When the other side closes, too, a *FIN* comes in, which is acknowledged. Now both sides are closed, but TCP waits a time equal to the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the timer goes off, TCP deletes the connection record.

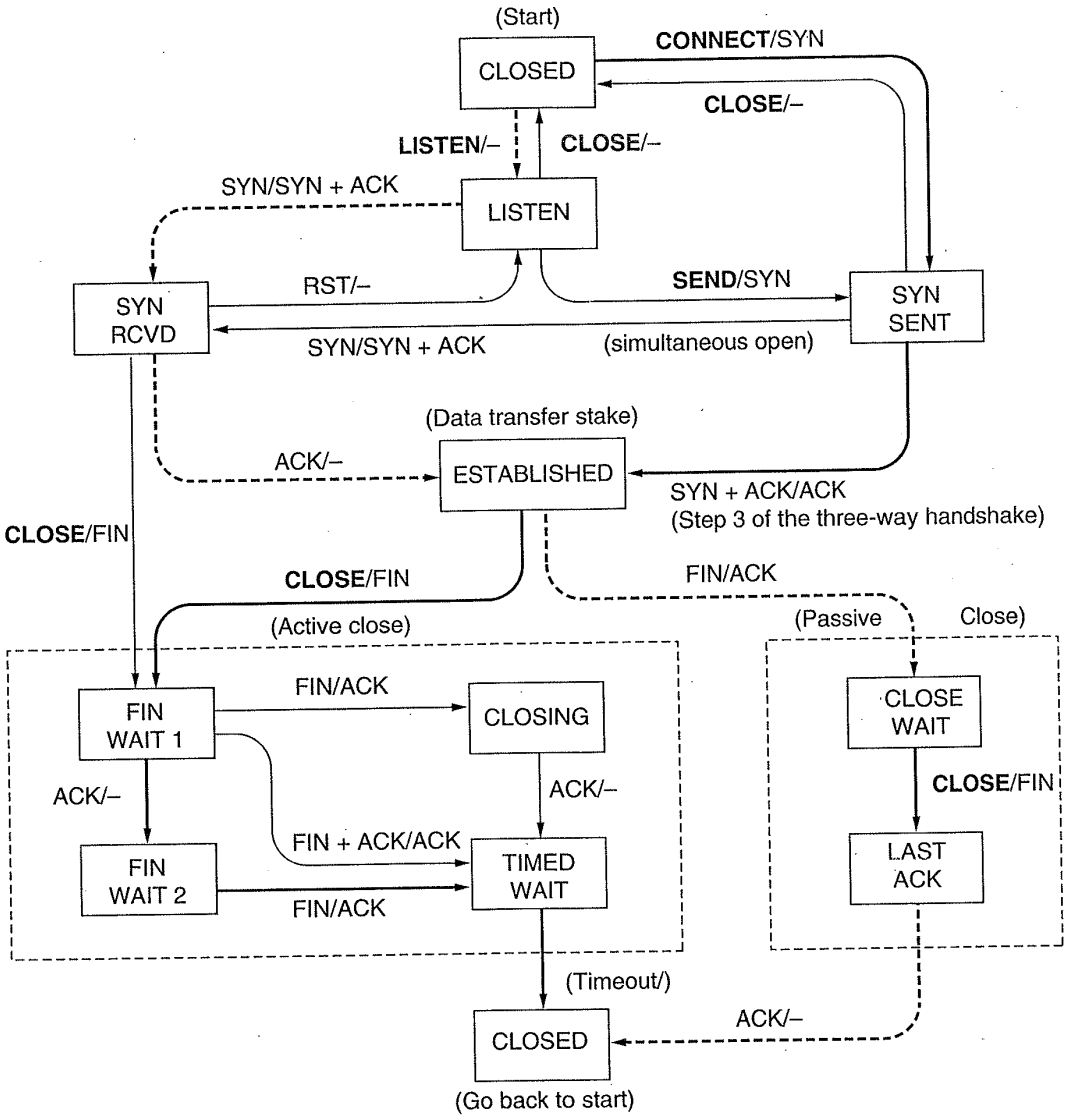


Fig. 6-28. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events.

Now let us examine connection management from the server's viewpoint. The server does a LISTEN and settles down to see who turns up. When a *SYN* comes in, it is acknowledged and the server goes to the *SYN RCVD* state. When the server's *SYN* is itself acknowledged, the three-way handshake is complete and the server goes to the *ESTABLISHED* state. Data transfer can now occur.

When the client has had enough, it does a CLOSE, which causes a *FIN* to arrive at the server (dashed box marked passive close). The server is then

signaled. When it, too, does a CLOSE, a *FIN* is sent to the client. When the client's acknowledgement shows up, the server releases the connection and deletes the connection record.

6.4.5. TCP Transmission Policy

Window management in TCP is not directly tied to acknowledgements as it is in most data link protocols. For example, suppose the receiver has a 4096-byte buffer as shown in Fig. 6-29. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment. However, since it now has only 2048 of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.

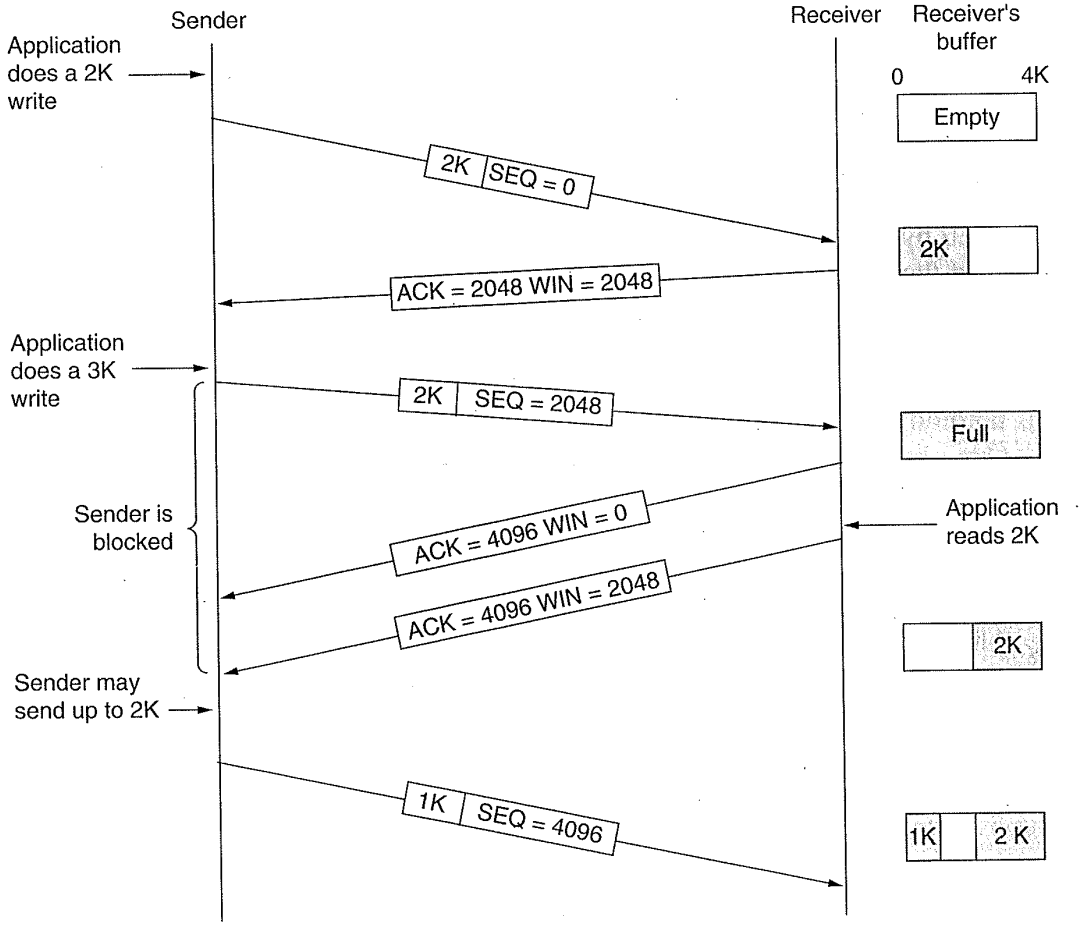


Fig. 6-29. Window management in TCP.

Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is 0. The sender must stop until the application process on

the receiving host has removed some data from the buffer, at which time TCP can advertise a larger window.

When the window is 0, the sender may not normally send segments, with two exceptions. First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine. Second, the sender may send a 1-byte segment to make the receiver reannounce the next byte expected and window size. The TCP standard explicitly provides this option to prevent deadlock if a window announcement ever gets lost.

Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible. For example, in Fig. 6-29, When the first 2 KB of data came in, TCP, knowing that it had a 4-KB window available, would have been completely correct in just buffering the data until another 2 KB came in, to be able to transmit a segment with a 4-KB payload. This freedom can be exploited to improve performance.

Consider a TELNET connection to an interactive editor that reacts on every keystroke. In the worst case, when a character arrives at the sending TCP entity, TCP creates a 21-byte TCP segment, which it gives to IP to send as a 41-byte IP datagram. At the receiving side, TCP immediately sends a 40-byte acknowledgement (20 bytes of TCP header and 20 bytes of IP header). Later, when the editor has read the byte, TCP sends a window update, moving the window 1 byte to the right. This packet is also 40 bytes. Finally, when the editor has processed the character, it echoes it as a 41-byte packet. In all, 162 bytes of bandwidth are used and four segments are sent for each character typed. When bandwidth is scarce, this method of doing business is not desirable.

One approach that many TCP implementations use to optimize this situation is to delay acknowledgements and window updates for 500 msec in the hope of acquiring some data on which to hitch a free ride. Assuming the editor echoes within 500 msec, only one 41-byte packet now need be sent back to the remote user, cutting the packet count and bandwidth usage in half.

Although this rule reduces the load placed on the network by the receiver, the sender is still operating inefficiently by sending 41-byte packets containing 1 byte of data. A way to reduce this usage is known as **Nagle's algorithm** (Nagle, 1984). What Nagle suggested is simple: when data come into the sender one byte at a time, just send the first byte and buffer all the rest until the outstanding byte is acknowledged. Then send all the buffered characters in one TCP segment and start buffering again until they are all acknowledged. If the user is typing quickly and the network is slow, a substantial number of characters may go in each segment, greatly reducing the bandwidth used. The algorithm additionally allows a new packet to be sent if enough data have trickled in to fill half the window or a maximum segment.

Nagle's algorithm is widely used by TCP implementations, but there are times when it is better to disable it. In particular, when an X-Windows application is

being run over the Internet, mouse movements have to sent to the remote computer. Gathering them up to send in bursts makes the mouse cursor move erratically, which makes for unhappy users.

Another problem that can ruin TCP performance is the **silly window syndrome** (Clark, 1982). This problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data 1 byte at a time. To see the problem, look at Fig. 6-30. Initially, the TCP buffer on the receiving side is full and the sender knows this (i.e., has a window of size 0). Then the interactive application reads one character from the TCP stream. This action makes the receiving TCP happy, so it sends a window update to the sender saying that it is all right to send 1 byte. The sender obliges and sends 1 byte. The buffer is now full, so the receiver acknowledges the 1-byte segment but sets the window to 0. This behavior can go on forever.

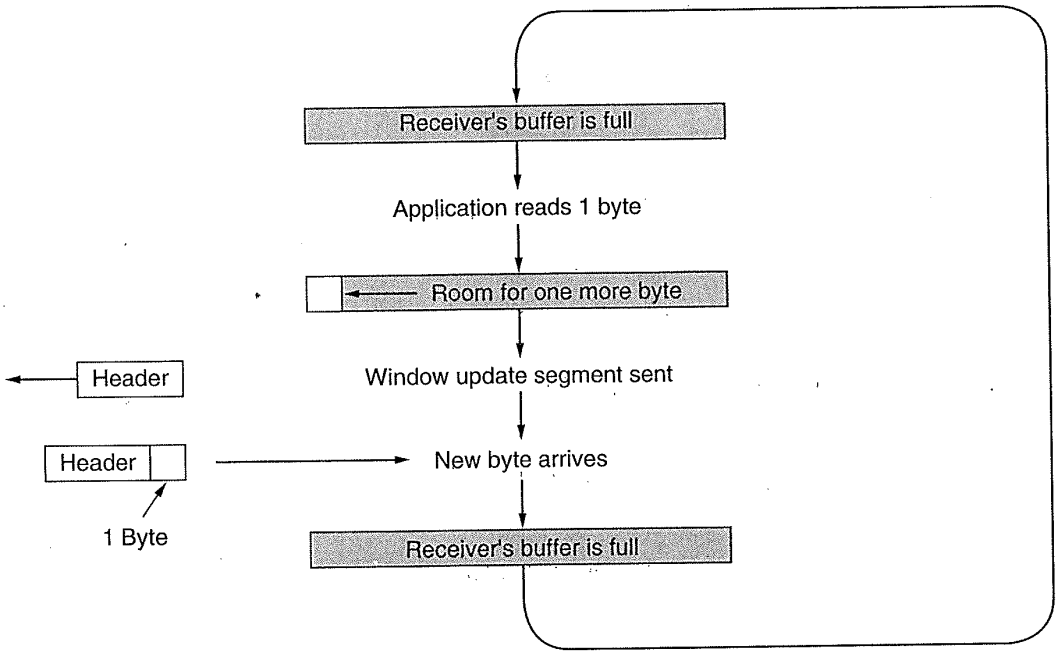


Fig. 6-30. Silly window syndrome.

Clark's solution is to prevent the receiver from sending a window update for 1 byte. Instead it is forced to wait until it has a decent amount of space available and advertise that instead. Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established, or its buffer is half empty, whichever is smaller.

Furthermore, the sender can also help by not sending tiny segments. Instead, it should try to wait until it has accumulated enough space in the window to send a full segment or at least one containing half of the receiver's buffer size (which it must estimate from the pattern of window updates it has received in the past).

Nagle's algorithm and Clark's solution to the silly window syndrome are complementary. Nagle was trying to solve the problem caused by the sending application delivering data to TCP a byte at a time. Clark was trying to solve the problem of the receiving application sucking the data up from TCP a byte at a time. Both solutions are valid and can work together. The goal is for the sender not to send small segments and the receiver not to ask for them.

The receiving TCP can go further in improving performance than just doing window updates in large units. Like the sending TCP, it also has the ability to buffer data, so it can block a READ request from the application until it has a large chunk of data to provide. Doing this reduces the number of calls to TCP, and hence the overhead. Of course, it also increases the response time, but for noninteractive applications like file transfer, efficiency may outweigh response time to individual requests.

Another receiver issue is what to do with out of order segments. They can be kept or discarded, at the receiver's discretion. Of course, acknowledgements can be sent only when all the data up to the byte acknowledged have been received. If the receiver gets segments 0, 1, 2, 4, 5, 6, and 7, it can acknowledge everything up to and including the last byte in segment 2. When the sender times out, it then retransmits segment 3. If the receiver has buffered segments 4 through 7, upon receipt of segment 3 it can acknowledge all bytes up to the end of segment 7.

6.4.6. TCP Congestion Control

When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. In this section we will discuss algorithms that have been developed over the past decade to deal with congestion. Although the network layer also tries to manage congestion, most of the heavy lifting is done by TCP because the real solution to congestion is to slow down the data rate.

In theory, congestion can be dealt with by employing a principle borrowed from physics: the law of conservation of packets. The idea is not to inject a new packet into the network until an old one leaves (i.e., is delivered). TCP attempts to achieve this goal by dynamically manipulating the window size.

The first step in managing congestion is detecting it. In the old days, detecting congestion was difficult. A timeout caused by a lost packet could have been caused by either (1) noise on a transmission line or (2) packet discard at a congested router. Telling the difference was difficult.

Nowadays, packet loss due to transmission errors is relatively rare because most long-haul trunks are fiber (although wireless networks are a different story). Consequently, most transmission timeouts on the Internet are due to congestion. All the Internet TCP algorithms assume that timeouts are caused by congestion and monitor timeouts for signs of trouble the way miners watch their canaries.

Before discussing how TCP reacts to congestion, let us first describe what it does to try to prevent it from occurring in the first place. When a connection is

established, a suitable window size has to be chosen. The receiver can specify a window based on its buffer size. If the sender sticks to this window size, problems will not occur due to buffer overflow at the receiving end, but they may still occur due to internal congestion within the network.

In Fig. 6-31, we see this problem illustrated hydraulically. In Fig. 6-31(a), we see a thick pipe leading to a small-capacity receiver. As long as the sender does not send more water than the bucket can contain, no water will be lost. In Fig. 6-31(b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case by overflowing the funnel).

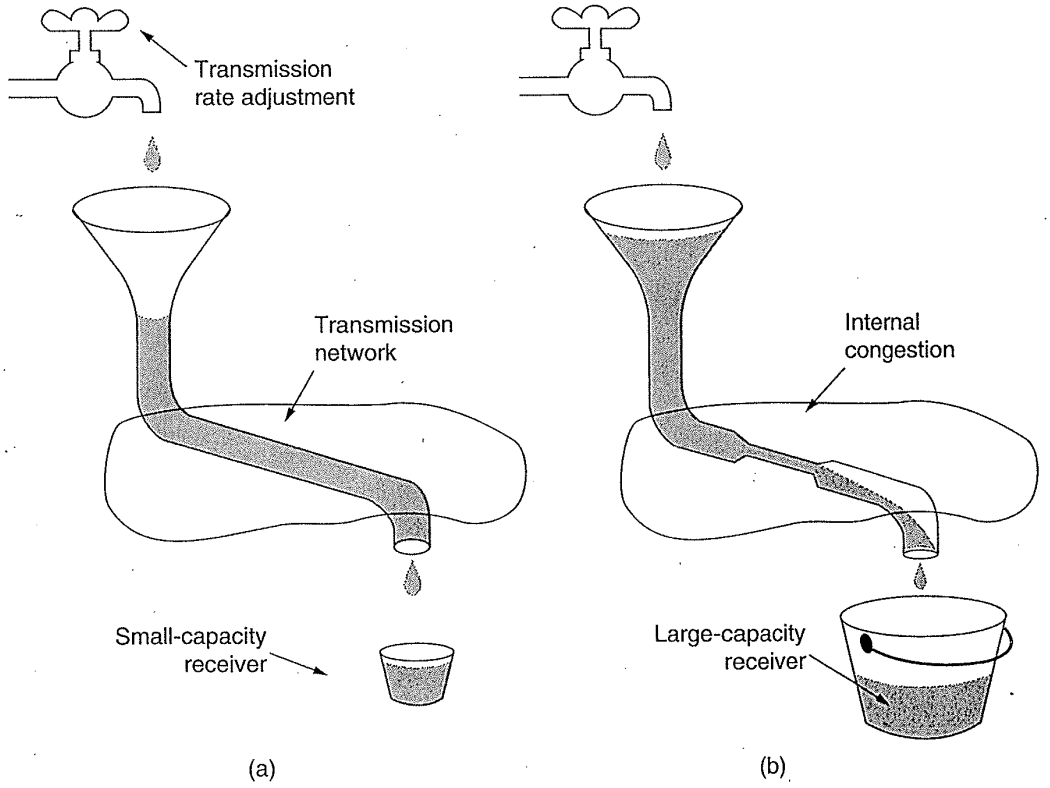


Fig. 6-31. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

The Internet solution is to realize that two potential problems exist—network capacity and receiver capacity—and to deal with each of them separately. To do so, each sender maintains two windows: the window the receiver has granted and a second window, the **congestion window**. Each reflects the number of bytes the sender may transmit. The number of bytes that may be sent is the minimum of the two windows. Thus the effective window is the minimum of what the sender

thinks is all right and what the receiver thinks is all right. If the receiver says "Send 8K" but the sender knows that bursts of more than 4K clog the network up, it sends 4K. On the other hand, if the receiver says "Send 8K" and the sender knows that bursts of up to 32K get through effortlessly, it sends the full 8K requested.

When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection. It then sends one maximum segment. If this segment is acknowledged before the timer goes off, it adds one segment's worth of bytes to the congestion window to make it two maximum size segments and sends two segments. As each of these segments is acknowledged, the congestion window is increased by one maximum segment size. When the congestion window is n segments, if all n are acknowledged on time, the congestion window is increased by the byte count corresponding to n segments. In effect, each burst successfully acknowledged doubles the congestion window.

The congestion window keeps growing exponentially until either a timeout occurs or the receiver's window is reached. The idea is that if bursts of size, say, 1024, 2048, and 4096 bytes work fine, but a burst of 8192 bytes gives a timeout, the congestion window should be set to 4096 to avoid congestion. As long as the congestion window remains at 4096, no bursts longer than that will be sent, no matter how much window space the receiver grants. This algorithm is called **slow start**, but it is not slow at all (Jacobson, 1988). It is exponential. All TCP implementations are required to support it.

Now let us look at the Internet congestion control algorithm. It uses a third parameter, the **threshold**, initially 64K, in addition to the receiver and congestion windows. When a timeout occurs, the threshold is set to half of the current congestion window, and the congestion window is reset to one maximum segment. Slow start is then used to determine what the network can handle, except that exponential growth stops when the threshold is hit. From that point on, successful transmissions grow the congestion window linearly (by one maximum segment for each burst) instead of one per segment. In effect, this algorithm is guessing that it is probably acceptable to cut the congestion window in half, and then it gradually works its way up from there.

As an illustration of how the congestion algorithm works, see Fig. 6-32. The maximum segment size here is 1024 bytes. Initially the congestion window was 64K, but a timeout occurred, so the threshold is set to 32K and the congestion window to 1K for transmission 0 here. The congestion window then grows exponentially until it hits the threshold (32K). Starting then it grows linearly.

Transmission 13 is unlucky (it should have known) and a timeout occurs. The threshold is set to half the current window (by now 40K, so half is 20K) and slow start initiated all over again. When the acknowledgements from transmission 18 start coming in, the first four each increment the congestion window by one segment, but after that, growth becomes linear again.

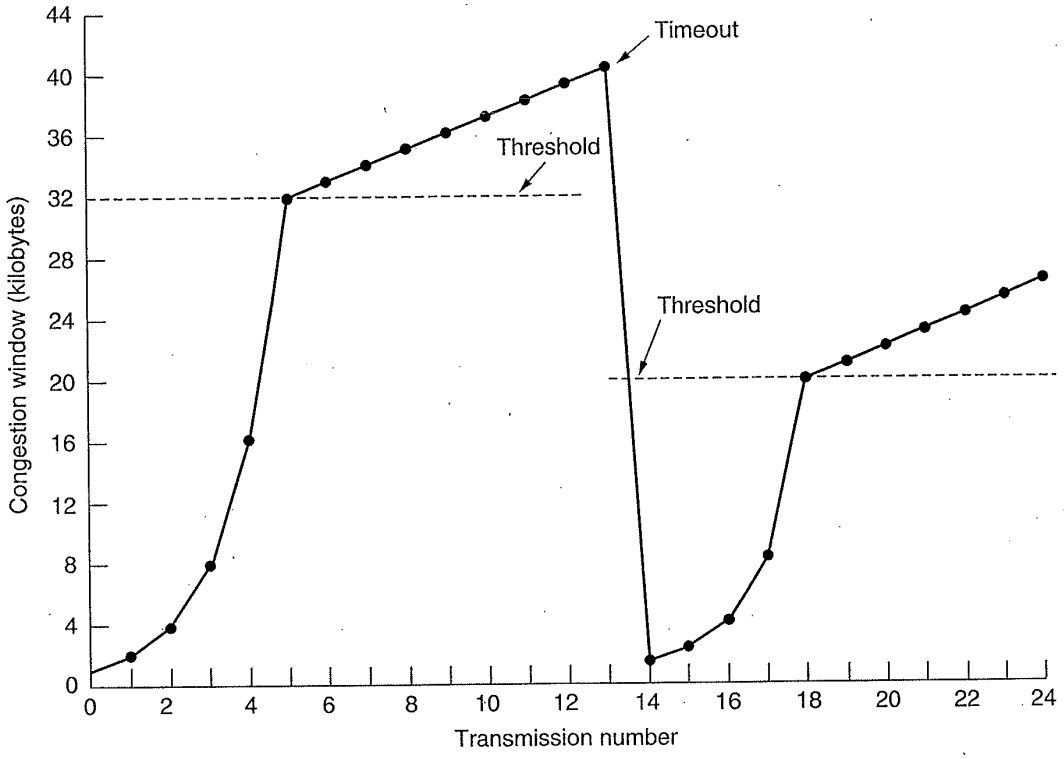


Fig. 6-32. An example of the Internet congestion algorithm.

If no more timeouts occur, the congestion window will continue to grow up to the size of the receiver's window. At that point, it will stop growing and remain constant as long as there are no more timeouts and the receiver's window does not change size. As an aside, if an ICMP SOURCE QUENCH packet comes in and is passed to TCP, this event is treated the same way as a timeout.

Work on improving the congestion control mechanism is continuing. For example, Brakmo et al. (1994) have reported improving TCP throughput by 40 percent to 70 percent by managing the clock more accurately, predicting congestion before timeouts occur, and using this early warning system to improve the slow start algorithm.

6.4.7. TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the **retransmission timer**. When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer started again). The question that arises is: How long should the timeout interval be?

This problem is much more difficult in the Internet transport layer than in the generic data link protocols of Chap. 3. In the latter case, the expected delay is highly predictable (i.e., has a low variance), so the timer can be set to go off just slightly after the acknowledgement is expected, as shown in Fig. 6-33(a). Since acknowledgements are rarely delayed in the data link layer, the absence of an acknowledgement at the expect time generally means the frame or the acknowledgement has been lost.

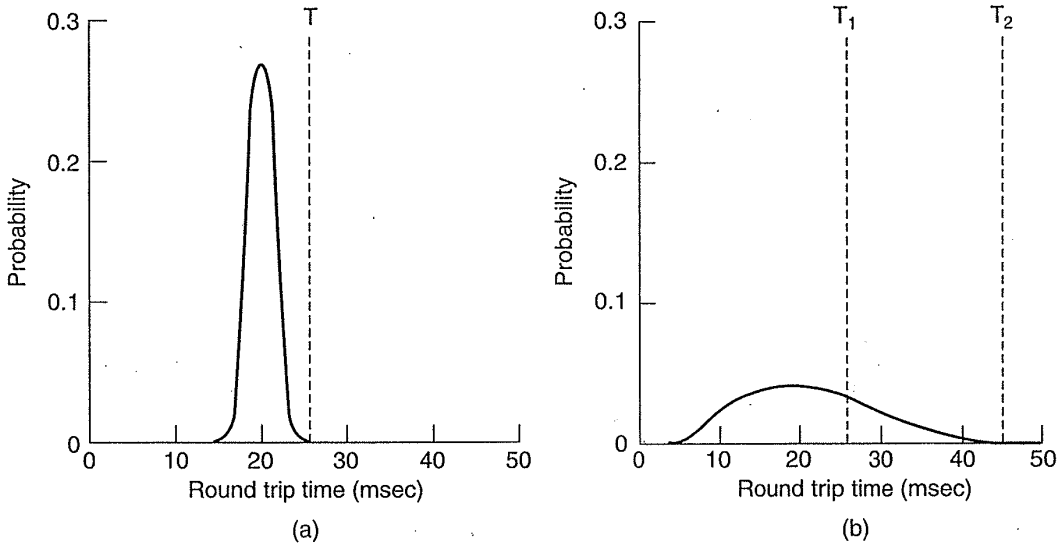


Fig. 6-33. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

TCP is faced with a radically different environment. The probability density function for the time it takes for a TCP acknowledgement to come back looks more like Fig. 6-33(b) than Fig. 6-33(a). Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say T_1 in Fig. 6-33(b), unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long, (T_2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved.

The solution is to use a highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurements of network performance. The algorithm generally used by TCP is due to Jacobson (1988) and works as follows. For each connection, TCP maintains a variable, *RTT*, that is the best current estimate of the round-trip time to the destination in question. When a segment is sent, a timer is started, both to see how long the acknowledgement takes and to

trigger a retransmission if it takes too long. If the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say, *M*. It then updates *RTT* according to the formula

$$RTT = \alpha RTT + (1 - \alpha)M$$

where α is a smoothing factor that determines how much weight is given to the old value. Typically $\alpha = 7/8$.

Even given a good value of *RTT*, choosing a suitable retransmission timeout is a nontrivial matter. Normally, TCP uses βRTT, but the trick is choosing β. In the initial implementations, β was always 2, but experience showed that a constant value was inflexible because it failed to respond when the variance went up.

In 1988, Jacobson proposed making β roughly proportional to the standard deviation of the acknowledgement arrival time probability density function so a large variance means a large β and vice versa. In particular, he suggested using the *mean deviation* as a cheap estimator of the *standard deviation*. His algorithm requires keeping track of another smoothed variable, *D*, the deviation. Whenever an acknowledgement comes in, the difference between the expected and observed values, $|RTT - M|$ is computed. A smoothed value of this is maintained in *D* by the formula

$$D = \alpha D + (1 - \alpha)\ |RTT - M|$$

where α may or may not be the same value used to smooth *RTT*. While *D* is not exactly the same as the standard deviation, it is good enough and Jacobson showed how it could be computed using only integer adds, subtracts, and shifts, a big plus. Most TCP implementations now use this algorithm and set the timeout interval to

$$\text{Timeout} = RTT + 4*D$$

The choice of the factor 4 is somewhat arbitrary, but it has two advantages. First, multiplication by 4 can be done with a single shift. Second, it minimizes unnecessary timeouts and retransmissions because less than one percent of all packets come in more than four standard deviations late. (Actually, Jacobson initially said to use 2, but later work has shown that 4 gives better performance.)

One problem that occurs with the dynamic estimation of *RTT* is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the estimate of *RTT*. Phil Karn discovered this problem the hard way. He is an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium (on a good day, half the packets get through). He made a simple proposal: do not update *RTT* on any segments that have been retransmitted. Instead, the timeout is doubled on each failure until the segments get through the first time. This fix is called **Karn's algorithm**. Most TCP implementations use it.

The retransmission timer is not the only one TCP uses. A second timer is the **persistence timer**. It is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, but the packet with the update is lost. Now both the sender and the receiver are waiting for each other to do something. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still zero, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent.

A third timer that some implementations use is the **keepalive timer**. When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check if the other side is still there. If it fails to respond, the connection is terminated. This feature is controversial because it adds overhead and may terminate an otherwise healthy connection due to a transient network partition.

The last timer used on each TCP connection is the one used in the *TIMED WAIT* state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

6.4.8. UDP

The Internet protocol suite also supports a connectionless transport protocol, **UDP (User Data Protocol)**. UDP provides a way for applications to send encapsulated raw IP datagrams and send them without having to establish a connection. Many client-server applications that have one request and one response use UDP rather than go to the trouble of establishing and later releasing a connection. UDP is described in RFC 768.

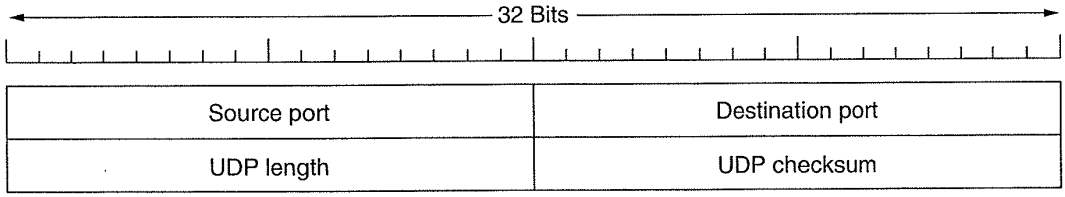


Fig. 6-34. The UDP header.

A UDP segment consists of an 8-byte header followed by the data. The header is shown in Fig. 6-34. The two ports serve the same function as they do in TCP: to identify the end points within the source and destination machines. The *UDP length* field includes the 8-byte header and the data. The *UDP checksum* includes the same format pseudoheader shown in Fig. 6-25, the UDP header, and the UDP data, padded out to an even number of bytes if need be. It is optional and stored as 0 if not computed (a true computed 0 is stored as all 1s, which is the same in 1's complement). Turning it off is foolish unless the quality of the data does not matter (e.g., digitized speech).

6.4.9. Wireless TCP and UDP

In theory, transport protocols should be independent of the technology of the underlying network layer. In particular, TCP should not care whether IP is running over fiber or over radio. In practice, it does matter because most TCP implementations have been carefully optimized based on assumptions that are true for wired networks but which fail for wireless networks. Ignoring the properties of wireless transmission can lead to a TCP implementation that is logically correct but has horrendous performance.

The principal problem is the congestion control algorithm. Nearly all TCP implementations nowadays assume that timeouts are caused by congestion, not by lost packets. Consequently, when a timer goes off, TCP slows down and sends less vigorously (e.g., Jacobson's slow start algorithm). The idea behind this approach is to reduce the network load and thus alleviate the congestion.

Unfortunately, wireless transmission links are highly unreliable. They lose packets all the time. The proper approach to dealing with lost packets is to send them again, and as quickly as possible. Slowing down just makes matters worse. If, say, 20 percent of all packets are lost, then when the sender transmits 100 packets/sec, the throughput is 80 packets/sec. If the sender slows down to 50 packets/sec, the throughput drops to 40 packets/sec.

In effect, when a packet is lost on a wired network, the sender should slow down. When one is lost on a wireless network, the sender should try harder. When the sender does not know what the network is, it is difficult to make the correct decision.

Frequently, the path from sender to receiver is inhomogeneous. The first 1000 km might be over a wired network, but the last 1 km might be wireless. Now making the correct decision on a timeout is even harder, since it matters where the problem occurred. A solution proposed by Bakne and Badrinath (1995), **indirect TCP**, is to split the TCP connection into two separate connections, as shown in Fig. 6-35. The first connection goes from the sender to the base station. The second one goes from the base station to the receiver. The base station simply copies packets between the connections in both directions.

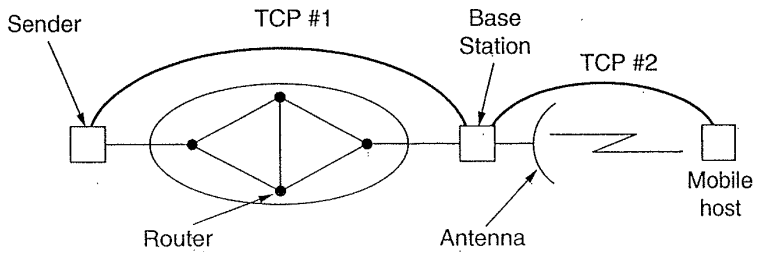


Fig. 6-35. Splitting a TCP connection into two connections.

The advantage of this scheme is that both connections are now homogeneous. Timeouts on the first connection can slow the sender down, whereas timeouts on the second one can speed it up. Other parameters can also be tuned separately for the two connections. The disadvantage is that it violates the semantics of TCP. Since each part of the connection is a full TCP connection, the base station acknowledges each TCP segment in the usual way. Only now, receipt of an acknowledgement by the sender does not mean that the receiver got the segment, only that the base station got it.

A different solution, due to Balakrishnan et al. (1995), does not break the semantics of TCP. It works by making several small modifications to the network layer code in the base station. One of the changes is the addition of a snooping agent that observes and caches TCP segments going out to the mobile host, and acknowledgements coming back from it. When the snooping agent sees a TCP segment going out to the mobile host but does not see an acknowledgement coming back before its (relatively short) timer goes off, it just retransmits that segment, without telling the source that it is doing so. It also generates a retransmission when it sees duplicate acknowledgements from the mobile host go by, invariably meaning that the mobile host has missed something. Duplicate acknowledgements are discarded on the spot, to avoid having the source misinterpret them as a sign of congestion.

One disadvantage of this transparency, however, is that if the wireless link is very lossy, the source may time out waiting for an acknowledgement and invoke the congestion control algorithm. With indirect TCP, the congestion control algorithm will never be started unless there really is congestion in the wired part of the network.

The Balakrishnan et al. paper also has a solution to the problem of lost segments originating at the mobile host. When the base station notices a gap in the inbound sequence numbers, it generates a request for a selective repeat of the missing bytes using a TCP option. Using these two fixes, the wireless link is made more reliable in both directions, without the source knowing about it, and without changing the semantics of TCP.

While UDP does not suffer from the same problems as TCP, wireless communication also introduces difficulties for it. The main trouble is that programs use UDP expecting it to be highly reliable. They know that no guarantees are given, but they still expect it to be near perfect. In a wireless environment, it will be far from perfect. For programs that are able to recover from lost UDP messages, but only at considerable cost, suddenly going from an environment where messages theoretically can be lost but rarely are, to one in which they are constantly being lost can result in a performance disaster.

Wireless communication also affects areas other than just performance. For example, how does a mobile host find a local printer to connect to, rather than use its home printer? Somewhat related to this is how to get the WWW page for the local cell, even if its name is not known. Also, WWW page designers tend to

assume lots of bandwidth is available. Putting a large logo on every page becomes counterproductive if it is going to take 30 sec to transmit at 9600 bps every time the page is referenced, irritating the users no end.

6.5. THE ATM AAL LAYER PROTOCOLS

It is not really clear whether or not ATM has a transport layer. On the one hand, the ATM layer has the functionality of a network layer, and there is another layer on top of it (AAL), which sort of makes AAL a transport layer. Some experts agree with this view (e.g., De Prycker, 1993, page 112). One of the protocols used here (AAL 5) is functionally similar to UDP, which is unquestionably a transport protocol.

On the other hand, none of the AAL protocols provide a reliable end-to-end connection, as TCP does (although with only very minor changes they could). Also, in most applications another transport layer is used on top of AAL. Rather than split hairs, we will discuss the AAL layer and its protocols in this chapter without making a claim that it is a true transport layer.

The AAL layer in ATM networks is radically different than TCP, largely because the designers were primarily interested in transmitting voice and video streams, in which rapid delivery is more important than accurate delivery. Remember that the ATM layer just outputs 53-byte cells one after another. It has no error control, no flow control, and no other control. Consequently, it is not well matched to the requirements that most applications need.

To bridge this gap, in Recommendation I.363, ITU has defined an end-to-end layer on top of the ATM layer. This layer, called **AAL (ATM Adaptation Layer)** has a tortuous history, full of mistakes, revisions, and unfinished business. In the following sections we will look at it and its design.

The goal of AAL is to provide useful services to application programs and to shield them from the mechanics of chopping data up into cells at the source and reassembling them at the destination. When ITU began defining AAL, it realized that different applications had different requirements, so it organized the service space along three axes:

1. Real-time service versus nonreal-time service.
2. Constant bit rate service versus variable bit rate service.
3. Connection-oriented service versus connectionless service.

In principle, with three axes and two values on each axis, eight distinct services can be defined, as shown in Fig. 6-36. ITU felt that only four of these were of any use, and named them classes A, B, C, and D, as noted. The others were not supported. Starting with ATM 4.0, Fig. 6-36 is somewhat obsolete, so it has been presented here mostly as background information to help understand why the

(Schneier, 1996). Chapter 8 of his email book does too (Schneier, 1995). Privacy and computers are also discussed in (Adam, 1995). These references are highly recommended for readers who wish to pursue their study of this subject.

7.2. DNS—Domain Name System

Programs rarely refer to hosts, mailboxes, and other resources by their binary network addresses. Instead of binary numbers, they use ASCII strings, such as *tana@art.ucsb.edu*. Nevertheless, the network itself only understands binary addresses, so some mechanism is required to convert the ASCII strings to network addresses. In the following sections we will study how this mapping is accomplished in the Internet.

Way back in the ARPANET, there was simply a file, *hosts.txt*, that listed all the hosts and their IP addresses. Every night, all the hosts would fetch it from the site at which it was maintained. For a network of a few hundred large timesharing machines, this approach worked reasonably well.

However, when thousands of workstations were connected to the net, everyone realized that this approach could not continue to work forever. For one thing, the size of the file would become too large. However, even more important, host name conflicts would occur constantly unless names were centrally managed, something unthinkable in a huge international network. To solve these problems, **DNS** (the **Domain Name System**) was invented.

The essence of DNS is the invention of a hierarchical, domain-based naming scheme and a distributed database system for implementing this naming scheme. It is primarily used for mapping host names and email destinations to IP addresses but can also be used for other purposes. DNS is defined in RFCs 1034 and 1035.

Very briefly, the way DNS is used is as follows. To map a name onto an IP address, an application program calls a library procedure called the **resolver**, passing it the name as a parameter. The resolver sends a UDP packet to a local DNS server, which then looks up the name and returns the IP address to the resolver, which then returns it to the caller. Armed with the IP address, the program can then establish a TCP connection with the destination, or send it UDP packets.

7.2.1. The DNS Name Space

Managing a large and constantly changing set of names is a nontrivial problem. In the postal system, name management is done by requiring letters to specify (implicitly or explicitly) the country, state or province, city, and street address of the addressee. By using this kind of hierarchical addressing, there is no confusion between the Marvin Anderson on Main St. in White Plains, N.Y. and the Marvin Anderson on Main St. in Austin, Texas. DNS works the same way.

Conceptually, the Internet is divided into several hundred top-level **domains**, where each domain covers many hosts. Each domain is partitioned into subdomains, and these are further partitioned, and so on. All these domains can be represented by a tree, as shown in Fig. 7-25. The leaves of the tree represent domains that have no subdomains (but do contain machines, of course) A leaf domain may contain a single host, or it may represent a company and contains thousands of hosts.

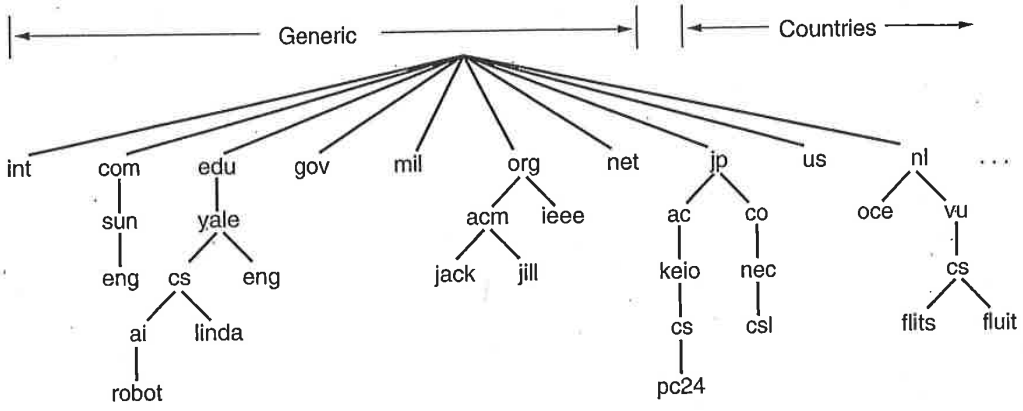


Fig. 7-25. A portion of the Internet domain name space.

The top-level domains come in two flavors: generic and countries. The generic domains are *com (commercial), edu* (educational institutions), *gov* (the U.S. federal government), *int* (certain international organizations), *mil* (the U.S. armed forces), *net* (network providers), and *org* (nonprofit organizations). The country domains include one entry for every country, as defined in ISO 3166.

Each domain is named by the path upward from it to the (unnamed) root. The components are separated by periods (pronounced "dot"). Thus Sun Microsystems engineering department might be *eng.sun.com.*, rather than a UNIX-style name such as */com/sun/eng*. Notice that this hierarchical naming means that *eng.sun.com.* does not conflict with a potential use of *eng* in *eng.yale.edu.*, which might be used by the Yale English department.

Domain names can be either absolute or relative. An absolute domain name ends with a period (e.g., *eng.sun.com.*), whereas a relative one does not. Relative names have to be interpreted in some context to uniquely determine their true meaning. In both cases, a named domain refers to a specific node in the tree and all the nodes under it.

Domain names are case insensitive, so *edu* and *EDU* mean the same thing. Component names can be up to 63 characters long, and full path names must not exceed 255 characters.

In principle, domains can be inserted into the tree in two different ways. For example, *cs.yale.edu* could equally well be listed under the *us* country domain as

cs.yale.ct.us. In practice, however, nearly all organizations in the United States are under a generic domain, and nearly all outside the United States are under the domain of their country. There is no rule against registering under two top-level domains, but doing so might be confusing, so few organizations do it.

Each domain controls how it allocates the domains under it. For example, Japan has domains *ac.jp* and *co.jp* that mirror *edu* and *com*. The Netherlands does not make this distinction and puts all organizations directly under *nl*. Thus all three of the following are university computer science departments:

1. *cs.yale.edu* (Yale University, in the United States)
2. *cs.vu.nl* (Vrije Universiteit, in The Netherlands)
3. *cs.keio.ac.jp* (Keio University, in Japan)

To create a new domain, permission is required of the domain in which it will be included. For example, if a VLSI group is started at Yale and wants to be known as *vlsi.cs.yale.edu*, it needs permission from whomever manages *cs.yale.edu*. Similarly, if a new university is chartered, say, the University of Northern South Dakota, it must ask the manager of the *edu* domain to assign it *unsd.edu*. In this way, name conflicts are avoided and each domain can keep track of all its subdomains. Once a new domain has been created and registered, it can create subdomains, such as *cs.unsd.edu*, without getting permission from anybody higher up the tree.

Naming follows organizational boundaries, not physical networks. For example, if the computer science and electrical engineering departments are located in the same building and share the same LAN, they can nevertheless have distinct domains. Similarly, even if computer science is split over Babbage Hall and Turing Hall, all the hosts in both buildings will normally belong to the same domain.

7.2.2. Resource Records

Every domain, whether it is a single host or a top-level domain, can have a set of **resource records** associated with it. For a single host, the most common resource record is just its IP address, but many other kinds of resource records also exist. When a resolver gives a domain name to DNS, what it gets back are the resource records associated with that name. Thus the real function of DNS is to map domain names onto resource records.

A resource record is a five-tuple. Although they are encoded in binary for efficiency, in most expositions resource records are presented as ASCII text, one line per resource record. The format we will use is as follows:

```
Domain_name  Time_to_live  Type  Class  Value
```

The *Domain_name* tells the domain to which this record applies. Normally, many records exist for each domain and each copy of the database holds information

about multiple domains. This field is thus the primary search key used to satisfy queries. The order of the records in the database is not significant. When a query is made about a domain, all the matching records of the class requested are returned.

The *Time_to_live* field gives an indication of how stable the record is. Information that is highly stable is assigned a large value, such as 86400 (the number of seconds in 1 day). Information that is highly volatile is assigned a small value, such as 60 (1 minute). We will come back to this point later when we have discussed caching.

The *Type* field tells what kind of record this is. The most important types are listed in Fig. 7-26.

Type	Meaning	Value
SOA	Start of Authority	Parameters for this zone
A	IP address of a host	32-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name Server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
HINFO	Host description	CPU and OS in ASCII
TXT	Text	Uninterpreted ASCII text

Fig. 7-26. The principal DNS resource record types.

An *SOA* record provides the name of the primary source of information about the name server's zone (described below), the email address of its administrator, a unique serial number, and various flags and timeouts.

The most important record type is the *A* (Address) record. It holds a 32-bit IP address for some host. Every Internet host must have at least one IP address, so other machines can communicate with it. Some hosts have two or more network connections, in which case they will have one type *A* resource record per network connection (and thus per IP address).

The next most important record type is the *MX* record. It specifies the name of the domain prepared to accept email for the specified domain. A common use of this record is to allow a machine that is not on the Internet to receive email from Internet sites. Delivery is accomplished by having the non-Internet site make an arrangement with some Internet site to accept email for it and forward it using whatever protocol the two of them agree on.

For example, suppose that Cathy is a computer science graduate student at UCLA. After she gets her degree in AI, she sets up a company, Electrobrain

Corporation, to commercialize her ideas. She cannot afford an Internet connection yet, so she makes an arrangement with UCLA to allow her to have her email sent there. A few times a day she will call up and collect it.

Next, she registers her company with the *com* domain and is assigned the domain *electrobrain.com*. She might then ask the administrator of the *com* domain to add an *MX* record to the *com* database as follows:

```
electrobrain.com   86400   IN   MX   1   mailserver.cs.ucla.edu
```

In this way, mail will be forwarded to UCLA where she can pick it up by logging in. Alternatively, UCLA could call her and transfer the email by any protocol they mutually agree on.

The *NS* records specify name servers. For example, every DNS database normally has an *NS* record for each of the top-level domains, so email can be sent to distant parts of the naming tree. We will come back to this point later.

CNAME records allow aliases to be created. For example, a person familiar with Internet naming in general wanting to send a message to someone whose login name is *paul* in the computer science department at M.I.T. might guess that *paul@cs.mit.edu* will work. Actually this address will not work, because the domain for M.I.T.'s computer science department is *lcs.mit.edu*. However, as a service to people who do not know this, M.I.T. could create a *CNAME* entry to point people and programs in the right direction. An entry like this one might do the job:

```
cs.mit.edu   86400   IN   CNAME   lcs.mit.edu
```

Like *CNAME*, *PTR* points to another name. However, unlike *CNAME*, which is really just a macro definition, *PTR* is a regular DNS datatype whose interpretation depends on the context in which it is found. In practice, it is nearly always used to associate a name with an IP address to allow lookups of the IP address and return the name of the corresponding machine.

HINFO records allow people to find out what kind of machine and operating system a domain corresponds to. Finally, *TXT* records allow domains to identify themselves in arbitrary ways. Both of these record types are for user convenience. Neither is required, so programs cannot count on getting them (and probably cannot deal with them if they do get them).

Getting back to the general structure of resource records, the fourth field of every resource record is the *Class*. For Internet information, it is always *IN*. For non-Internet information, other codes can be used.

Finally, we come to the *Value* field. This field can be a number, a domain name, or an ASCII string. The semantics depend on the record type. A short description of the *Value* fields for each of the principal records types is given in Fig. 7-26.

As an example of the kind of information one might find in the DNS database of a domain, see Fig. 7-27. This figure depicts part of a (semihypothetical)

database for the *cs.vu.nl* domain shown in Fig. 7-25. The database contains seven types of resource records.

```
; Authoritative data for cs.vu.nl
cs.vu.nl.        86400  IN  SOA     star boss (952771,7200,7200,2419200,86400)
cs.vu.nl.        86400  IN  TXT     "Faculteit Wiskunde en Informatica."
cs.vu.nl.        86400  IN  TXT     "Vrije Universiteit Amsterdam."
cs.vu.nl.        86400  IN  MX      1 zephyr.cs.vu.nl.
cs.vu.nl.        86400  IN  MX      2 top.cs.vu.nl.

flits.cs.vu.nl.  86400  IN  HINFO   Sun Unix
flits.cs.vu.nl.  86400  IN  A       130.37.16.112
flits.cs.vu.nl.  86400  IN  A       192.31.231.165
flits.cs.vu.nl.  86400  IN  MX      1 flits.cs.vu.nl.
flits.cs.vu.nl.  86400  IN  MX      2 zephyr.cs.vu.nl.
flits.cs.vu.nl.  86400  IN  MX      3 top.cs.vu.nl.
www.cs.vu.nl.    86400  IN  CNAME   star.cs.vu.nl
ftp.cs.vu.nl.    86400  IN  CNAME   zephyr.cs.vu.nl

rowboat                 IN  A       130.37.56.201
                        IN  MX      1 rowboat
                        IN  MX      2 zephyr
                        IN  HINFO   Sun Unix

little-sister           IN  A       130.37.62.23
                        IN  HINFO   Mac MacOS

laserjet                IN  A       192.31.231.216
                        IN  HINFO   "HP Laserjet IIISi" Proprietary
```

Fig. 7-27. A portion of a possible DNS database for *cs.vu.nl*

The first noncomment line of Fig. 7-27 gives some basic information about the domain, which will not concern us further. The next two lines give textual information about where the domain is located. Then come two entries giving the first and second places to try to deliver email sent to *person@cs.vu.nl*. The *zephyr* (a specific machine) should be tried first. If that fails, the *top* should be tried next.

After the blank line, added for readability, come lines telling that the *flits* is a Sun workstation running UNIX and giving both of its IP addresses. Then three choices are given for handling email sent to *flits.cs.vu.nl*. First choice is naturally the *flits* itself, but if it is down, the *zephyr* and *top* are the second and third choices. Next comes an alias, *www.cs.vu.nl*, so that this address can be used without designating a specific machine. Creating this alias allows *cs.vu.nl* to change its World Wide Web server without invalidating the address people use to get to it. A similar argument holds for *ftp.cs.vu.nl*.

The next four lines contain a typical entry for a workstation, in this case, *rowboat.cs.vu.nl*. The information provided contains the IP address, the primary and secondary mail drops, and information about the machine. Then comes an entry for a non-UNIX system that is not capable of receiving mail itself, followed by an entry for a laser printer.

What is not shown (and is not in this file), are the IP addresses to use to look up the top level domains. These are needed to look up distant hosts, but since they are not part of the *cs.vu.nl* domain, they are not in this file. They are supplied by the root servers, whose IP addresses are present in a system configuration file and loaded into the DNS cache when the DNS server is booted. They have very long timeouts, so once loaded, they are never purged from the cache.

7.2.3. Name Servers

In theory at least, a single name server could contain the entire DNS database and respond to all queries about it. In practice, this server would be so overloaded as to be useless. Furthermore, if it ever went down, the entire Internet would be crippled.

To avoid the problems associated with having only a single source of information, the DNS name space is divided up into nonoverlapping **zones**. One possible way to divide up the name space of Fig. 7-25 is shown in Fig. 7-28. Each zone contains some part of the tree and also contains name servers holding the authoritative information about that zone. Normally, a zone will have one primary name server, which gets its information from a file on its disk, and one or more secondary name servers, which get their information from the primary name server. To improve reliability, some servers for a zone can be located outside the zone.

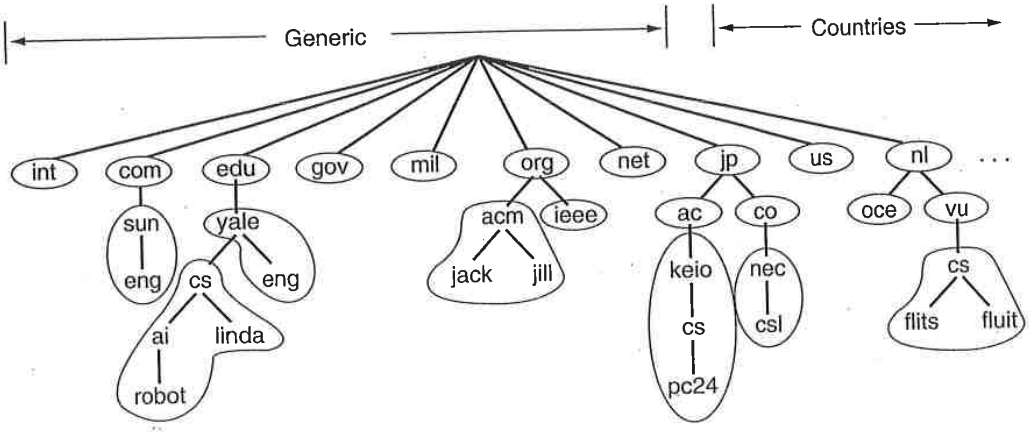


Fig. 7-28. Part of the DNS name space showing the division into zones.

Where the zone boundaries are placed within a zone is up to that zone's administrator. This decision is made in large part based on how many name

servers are desired, and where. For example, in Fig. 7-28, Yale has a server for *yale.edu* that handles *eng.yale.edu* but not *cs.yale.edu*, which is a separate zone with its own name servers. Such a decision might be made when a department such as English does not wish to run its own name server, but a department such as computer science does. Consequently, *cs.yale.edu* is a separate zone but *eng.yale.edu* is not.

When a resolver has a query about a domain name, it passes the query to one of the local name servers. If the domain being sought falls under the jurisdiction of the name server, such as *ai.cs.yale.edu* falling under *cs.yale.edu*, it returns the authoritative resource records. An **authoritative record** is one that comes from the authority that manages the record, and is thus always correct. Authoritative records are in contrast to cached records, which may be out of date.

If, however, the domain is remote and no information about the requested domain is available locally, the name server sends a query message to the top-level name server for the domain requested. To make this process clearer, consider the example of Fig. 7-29. Here, a resolver on *flits.cs.vu.nl* wants to know the IP address of the host *linda.cs.yale.edu*. In step 1, it sends a query to the local name server, *cs.vu.nl*. This query contains the domain name sought, the type (*A*) and the class (*IN*).

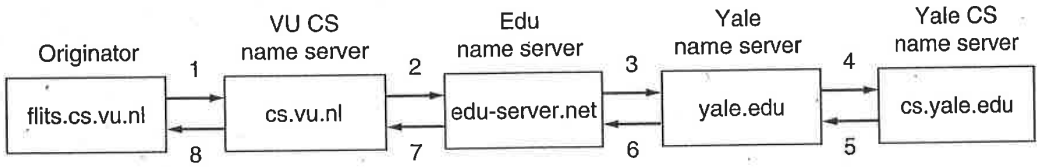


Fig. 7-29. How a resolver looks up a remote name in eight steps.

Let us suppose the local name server has never had a query for this domain before and knows nothing about it. It may ask a few other nearby name servers, but if none of them know, it sends a UDP packet to the server for *edu* given in its database (see Fig. 7-29), *edu-server.net*. It is unlikely that this server knows the address of *linda.cs.yale.edu*, and probably does not know *cs.yale.edu* either, but it must know all of its own children, so it forwards the request to the name server for *yale.edu* (step 3). In turn, this one forwards the request to *cs.yale.edu* (step 4), which must have the authoritative resource records. Since each request is from a client to a server, the resource record requested works its way back in steps 5 through 8.

Once these records get back to the *cs.vu.nl* name server, they will be entered into a cache there, in case they are needed later. However, this information is not authoritative, since changes made at *cs.yale.edu* will not be propagated to all the caches in the world that may know about it. For this reason, cache entries should not live too long. This is the reason that the *Time_to_live* field is included in each resource record. It tells remote name servers how long to cache records. If a

certain machine has had the same IP address for years, it may be safe to cache that information for 1 day. For more volatile information, it might be safer to purge the records after a few seconds or a minute.

It is worth mentioning that the query method described here is known as a **recursive query**, since each server that does not have the requested information goes and finds it somewhere, then reports back. An alternative form is also possible. In this form, when a query cannot be satisfied locally, the query fails, but the name of the next server along the line to try is returned. This procedure gives the client more control over the search process. Some servers do not implement recursive queries and always return the name of the next server to try.

It is also worth pointing out that when a DNS client fails to get a response before its timer goes off, it normally will try another server next time. The assumption here is that the server is probably down, rather than the request or reply got lost.

7.3. SNMP—SIMPLE NETWORK MANAGEMENT PROTOCOL

In the early days of the ARPANET, if the delay to some host became unexpectedly large, the person detecting the problem would just run the Ping program to bounce a packet off the destination. By looking at the timestamps in the header of the packet returned, the location of the problem could usually be pinpointed and some appropriate action taken. In addition, the number of routers was so small, that it was feasible to ping each one to see if it was sick.

When the ARPANET turned into the worldwide Internet, with multiple backbones and multiple operators, this solution ceased to be adequate, so better tools for network management were needed. Two early attempts were defined in RFC 1028 and RFC 1067, but these were short lived. In May 1990, RFC 1157 was published, defining version 1 of **SNMP (Simple Network Management Protocol)**. Along with a companion document (RFC 1155) on management information, SNMP provided a systematic way of monitoring and managing a computer network. This framework and protocol were widely implemented in commercial products and became the de facto standards for network management.

As experience was gained, shortcomings in SNMP came to light, so an enhanced version of SNMP (SNMPv2) was defined (in RFCs 1441 to 1452) and started along the road to become an Internet standard. In the sections to follow, we will give a brief discussion of the SNMP (meaning SNMPv2) model and protocol.

Although SNMP was designed with the idea of its being simple, at least one author has managed to produce a 600-page book on it (Stallings, 1993a). For more compact descriptions (450-550 pages), see the books by Rose (1994) and Rose and McCloghrie (1995), both of whom were among the designers of SNMP. Other references are (Feit, 1995; and Hein and Griffiths, 1995).

actual protocol that the management station and agents speak. The protocol itself is defined in RFC 1448.

The normal way that SNMP is used is that the management station sends a request to an agent asking it for information or commanding it to update its state in a certain way. Ideally, the agent just replies with the requested information or confirms that it has updated its state as requested. Data are sent using the ASN.1 transfer syntax. However, various errors can also be reported, such as No Such Variable.

SNMP defines seven messages that can be sent. The six messages from an initiator are listed in Fig. 7-38 (the seventh message is the response message). The first three request variable values to be sent back. The first format names the variables it wants explicitly. The second one asks for the next variable, allowing a manager to step through the entire MIB alphabetically (the default is the first variable). The third is for large transfers, such as tables.

Message	Description
Get-request	Requests the value of one or more variables
Get-next-request	Requests the variable following this one
Get-bulk-request	Fetches a large table
Set-request	Updates one or more variables
Inform-request	Manager-to-manager message describing local MIB
SnmpV2-trap	Agent-to-manager trap report

Fig. 7-38. SNMP message types.

Then comes a message that allows the manager to update an agent's variables, to the extent that the object specification permits such updates, of course. Next is an informational request that allows one manager to tell another one which variables it is managing. Finally, comes the message sent from an agent to a manager when a trap has sprung.

7.4. ELECTRONIC MAIL

Having finished looking at some of the support protocols used in the application layer, we finally come to real applications. When asked: "What are you going to do now?" few people will say: "I am going to look up some names with DNS." People do say they are going to read their email or news, surf the Web, or watch a movie over the net. In the remainder of this chapter, we will explain in a fair amount of detail how these four applications work.

Electronic mail, or **email**, as it is known to its many fans, has been around for over two decades. The first email systems simply consisted of file transfer protocols, with the convention that the first line of each message (i.e., file) contained the recipient's address. As time went on, the limitations of this approach became more obvious. Some of the complaints were

1. Sending a message to a group of people was inconvenient. Managers often need this facility to send memos to all their subordinates.
2. Messages had no internal structure, making computer processing difficult. For example, if a forwarded message was included in the body of another message, extracting the forwarded part from the received message was difficult.
3. The originator (sender) never knew if a message arrived or not.
4. If someone was planning to be away on business for several weeks and wanted all incoming email to be handled by his secretary, this was not easy to arrange.
5. The user interface was poorly integrated with the transmission system requiring users first to edit a file, then leave the editor and invoke the file transfer program.
6. It was not possible to create and send messages containing a mixture of text, drawings, facsimile, and voice.

As experience was gained, more elaborate email systems were proposed. In 1982, the ARPANET email proposals were published as RFC 821 (transmission protocol) and RFC 822 (message format). These have since become the de facto Internet standards. Two years later, CCITT drafted its X.400 recommendation, which was later taken over as the basis for OSI's MOTIS. In 1988, CCITT modified X.400 to align it with MOTIS. MOTIS was to be the flagship application for OSI, a system that was to be all things to all people.

After a decade of competition, email systems based on RFC 822 are widely used, whereas those based on X.400 have disappeared under the horizon. How a system hacked together by a handful of computer science graduate students beat an official international standard strongly backed by all the PTTs worldwide, many governments, and a substantial part of the computer industry brings to mind the Biblical story of David and Goliath. The reason for RFC 822's success is not that it is so good, but that X.400 is so poorly designed and so complex that nobody could implement it well. Given a choice between a simple-minded, but working, RFC 822-based email system and a supposedly truly wonderful, but nonworking, X.400 email system, most organizations chose the former. For a long diatribe on what is wrong with X.400, see Appendix C of (Rose, 1993). Consequently, our discussion of email will focus on RFC 821 and RFC 822 as used in the Internet.

subtype allows each part to be different, with no additional structure imposed. In contrast, with the *alternative* subtype, each part must contain the same message but expressed in a different medium or encoding. For example, a message could be sent in plain ASCII, in richtext, and in PostScript. A properly-designed user agent getting such a message would display it in PostScript if possible. Second choice would be richtext. If neither of these were possible, the flat ASCII text would be displayed. The parts should be ordered from simplest to most complex to help recipients with pre-MIME user agents make some sense of the message (e.g., even a pre-MIME user can read flat ASCII text).

The *alternative* subtype can also be used for multiple languages. In this context, the Rosetta Stone can be thought of as an early *multipart/alternative* message.

A multimedia example is shown in Fig. 7-46. Here a birthday greeting is transmitted both as text and as a song. If the receiver has an audio capability, the user agent there will fetch the sound file, *birthday.snd*, and play it. If not, the lyrics are displayed on the screen in stony silence. The parts are delimited by two hyphens followed by the (user-defined) string specified in the *boundary* parameter.

Note that the *Content-Type* header occurs in three positions within this example. At the top level, it indicates that the message has multiple parts. Within each part, it gives the type and subtype of that part. Finally, within the body of the second part, it is required to tell the user agent what kind of an external file it is to fetch. To indicate this slight difference in usage, we have used lowercase letters here, although all headers are case insensitive. The *content-transfer-encoding* is similarly required for any external body that is not encoded as 7-bit ASCII.

Getting back to the subtypes for multipart messages, two more possibilities exist. The *parallel* subtype is used when all parts must be "viewed" simultaneously. For example, movies often have an audio channel and a video channel. Movies are more effective if these two channels are played back in parallel, instead of consecutively.

Finally, the *digest* subtype is used when many messages are packed together into a composite message. For example, some discussion groups on the Internet collect messages from subscribers and then send them out as a single *multipart/digest* message.

7.4.4. Message Transfer

The message transfer system is concerned with relaying messages from originator to the recipient. The simplest way to do this is to establish a transport connection from the source machine to the destination machine and then just transfer the message. After examining how this is normally done, we will examine some situations in which this does not work and what can be done about them.

```
From: elinor@abc.com
To: carolyn@xyz.com
MIME-Version: 1.0
Message-Id: <0704760941.AA00747@abc.com>
Content-Type: multipart/alternative; boundary=qwertyuiopasdfghjklzxcvbnm
Subject: Earth orbits sun integral number of times

This is the preamble. The user agent ignores it. Have a nice day.

--qwertyuiopasdfghjklzxcvbnm
Content-Type: text/richtext

Happy birthday to you
Happy birthday to you
Happy birthday dear <bold> Carolyn </bold>
Happy birthday to you

--qwertyuiopasdfghjklzxcvbnm
Content-Type: message/external-body;
        access-type="anon-ftp";
        site="bicycle.abc.com";
        directory="pub";
        name="birthday.snd"

content-type: audio/basic
content-transfer-encoding: base64
--qwertyuiopasdfghjklzxcvbnm--
```

Fig. 7-46. A multipart message containing richtext and audio alternatives.

SMTP—Simple Mail Transfer Protocol

Within the Internet, email is delivered by having the source machine establish a TCP connection to port 25 of the destination machine. Listening to this port is an email daemon that speaks **SMTP (Simple Mail Transfer Protocol)**. This daemon accepts incoming connections and copies messages from them into the appropriate mailboxes. If a message cannot be delivered, an error report containing the first part of the undeliverable message is returned to the sender.

SMTP is a simple ASCII protocol. After establishing the TCP connection to port 25, the sending machine, operating as the client, waits for the receiving machine, operating as the server, to talk first. The server starts by sending a line of text giving its identity and telling whether or not it is prepared to receive mail. If it is not, the client releases the connection and tries again later.

If the server is willing to accept email, the client announces whom the email is coming from and whom it is going too. If such a recipient exists at the

destination, the server gives the client the go-ahead to send the message. Then the client sends the message and the server acknowledges it. No checksums are generally needed because TCP provides a reliable byte stream. If there is more email, that is now sent. When all the email has been exchanged in both directions, the connection is released. A sample dialog for sending the message of Fig. 7-46, including the numerical codes used by SMTP, is shown in Fig. 7-47. The lines sent by the client are marked *C:*; those sent by the server are marked *S:*.

A few comments about Fig. 7-47 may be helpful. The first command from the client is indeed *HELO*. Of the two four-character abbreviations for *HELLO*, this one has numerous advantages over its competitor. Why all the commands had to be four characters has been lost in the mists of time.

In Fig. 7-47, the message is sent to only one recipient, so only one *RCPT* command is used. Multiple such commands are allowed to send a single message to multiple receivers. Each one is individually acknowledged or rejected. Even if some recipients are rejected (because they do not exist at the destination), the message can be sent to the remainder.

Finally, although the syntax of the four-character commands from the client is rigidly specified, the syntax of the replies is less rigid. Only the numerical code really counts. Each implementation can put whatever string it wants after the code.

Even though the SMTP protocol is well defined (by RFC 821), a few problems can still arise. One problem relates to message length. Some older implementations cannot handle messages exceeding 64KB. Another problem relates to timeouts. If the client and server have different timeouts, one of them may give up while the other is still busy, unexpectedly terminating the connection. Finally, in rare situations, infinite mailstorms can be triggered. For example, if host 1 holds mailing list *A* and host 2 holds mailing list *B* and each list contains an entry for the other one, then any message sent to either list will generate a never-ending amount of email traffic.

To get around some of these problems, extended STMP (**ESMTP**) has been defined in RFC 1425. Clients wanting to use it should send an *EHLO* message instead of *HELO* initially. If this is rejected, then the server is a regular SMTP server, and the client should proceed in the usual way. If the *EHLO* is accepted, then new commands and parameters are allowed. The standardization of these commands and parameters is an ongoing process.

Email Gateways

Email using SMTP works best when both the sender and the receiver are on the Internet and can support TCP connections between sender and receiver. However, many machines that are not on the Internet still want to send and receive email from Internet sites. For example, many companies intentionally do not

```
                    S: 220 xyz.com SMTP service ready
C: HELO abc.com
                    S: 250 xyz.com says hello to abc.com
C: MAIL FROM: <elinor@abc.com>
                    S: 250 sender ok
C: RCPT TO: <carolyn@xyz.com>
                    S: 250 recipient ok
C: DATA
                    S: 354 Send mail; end with "." on a line by itself
C: From: elinor@abc.com
C: To: carolyn@xyz.com
C: MIME-Version: 1.0
C: Message-Id: <0704760941.AA00747@abc.com>
C: Content-Type: multipart/alternative; boundary=qwertyuiopasdfghjklzxcvbnm
C: Subject: Earth orbits sun integral number of times
C:
C: This is the preamble. The user agent ignores it. Have a nice day.
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: text/richtext
C:
C: Happy birthday to you
C: Happy birthday to you
C: Happy birthday dear <bold> Carolyn </bold>
C: Happy birthday to you
C:
C: --qwertyuiopasdfghjklzxcvbnm
C: Content-Type: message/external-body;
C:        access-type="anon-ftp";
C:        site="bicycle.abc.com";
C:        directory="pub";
C:        name="birthday.snd"
C:
C: content-type: audio/basic
C: content-transfer-encoding: base64
C: --qwertyuiopasdfghjklzxcvbnm
C: .
                    S: 250 message accepted
C: QUIT
                    S: 221 xyz.com closing connection
```

Fig. 7-47. Transferring a message from *elinor@abc.com* to *carolyn@xyz.com*.

want to be on the Internet for security reasons. Some of them even remove themselves from the Internet by erecting firewalls between themselves and the Internet.

Another problem occurs when the sender speaks only RFC 822 and the

receiver speaks only X.400 or some proprietary vendor-specific mail protocol. Since all these worlds differ in message formats and protocols, direct communication is impossible.

Both of these problems are solved using application layer **email gateways**. In Fig. 7-48 host 1 speaks only TCP/IP and RFC 822, whereas host 2 speaks only OSI TP4 and X.400. Nevertheless, they can exchange email using an email gateway. The procedure is for host 1 to establish a TCP connection to the gateway and then use SMTP to transfer a message (1) there. The daemon on the gateway then puts the message in a buffer of messages destined for host 2. Later, a TP4 connection (the OSI equivalent to TCP) is established with host 2 and the message (2) is transferred using the OSI equivalent of SMTP. All the gateway process has to do is to extract incoming messages from one queue and deposit them in another.

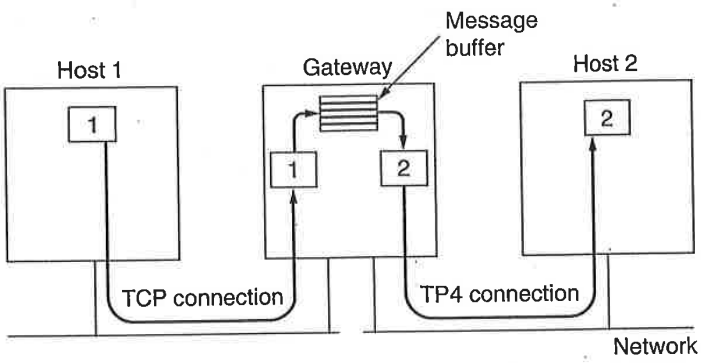


Fig. 7-48. Transferring email using an application layer email gateway.

It looks easy, but it is not. The first problem is that Internet addresses and X.400 addresses are totally different. An elaborate mapping mechanism is needed between them. The second problem is that envelope or header fields that are present in one system may not be present in the other. For example, if one system requires priority classes and the other does not have this concept at all, in one direction valuable information must be dropped and in the other it must be generated out of thin air.

An even worse concept is what to do if body parts are incompatible. What should a gateway do with a message from the Internet whose body holds a reference to an audio file to be obtained by FTP if the destination system does not support this concept? What should it do when an X.400 system tells it to deliver a message to a certain address, but if that fails, to send the contents by fax? Using fax is not part of the RFC 822 model. Clearly, there are no simple solutions here. For simple unstructured text messages in ASCII, gatewaying is a reasonable solution, but for anything fancier, the idea tends to break down.

Final Delivery

Up until now, we have assumed that all users work on machines that are capable of sending and receiving email. Frequently this situation is false. For example, at many companies, users work at desktop PCs that are not on the Internet and are not capable of sending or receiving email from outside the company. Instead, the company has one or more email servers that can send and receive email. To send or receive messages, a PC must talk to an email server using some kind of delivery protocol.

A simple protocol used for fetching email from a remote mailbox is **POP3** (**Post Office Protocol**), which is defined in RFC 1225. It has commands for the user to log in, log out, fetch messages, and delete messages. The protocol itself consists of ASCII text and has something of the flavor of SMTP. The point of POP3 is to fetch email from the remote mailbox and store it on the user's local machine to be read later.

A more sophisticated delivery protocol is **IMAP** (**Interactive Mail Access Protocol**), which is defined in RFC 1064. It was designed to help the user who uses multiple computers, perhaps a workstation in the office, a PC at home, and a laptop on the road. The basic idea behind IMAP is for the email server to maintain a central repository that can be accessed from any machine. Thus unlike POP3, IMAP does not copy email to the user's personal machine because the user may have several.

IMAP has many features, such as the ability to address mail not by arrival number as is done in Fig. 7-40, but by using attributes (e.g., Give me the first message from Sam). In this view, a mailbox is more like a relational database system than a linear sequence of messages.

Yet a third delivery protocol is **DMSP** (**Distributed Mail System Protocol**), which is part of the PCMAIL system and described in RFC 1056. This one does not assume that all email is on one server, as do POP3 and IMAP. Instead, it allows users to download email from the server to a workstation, PC, or laptop and then disconnect. The email can be read and answered while disconnected. When reconnection occurs later, email is transferred and the system is resynchronized.

Independent of whether email is delivered directly to the user's workstation or to a remote server, many systems provide hooks for additional processing of incoming email. An especially valuable tool for many email users is the ability to set up **filters**. These are rules that are checked when email comes in or when the user agent is started. Each rule specifies a condition and an action. For example, a rule could say that any message from Andrew S. Tanenbaum should be displayed in a 24-point flashing red boldface font (or alternatively, be discarded automatically without comment).

Another delivery feature often provided is the ability to (temporarily) forward incoming email to a different address. This address can even be a computer

operated by a commercial paging service, which then pages the user by radio or satellite, displaying the *Subject:* line on his beeper.

Still another common feature of final delivery is the ability to install a **vacation daemon**. This is a program that examines each incoming message and sends the sender an insipid reply such as

Hi. I'm on vacation. I'll be back on the 24th of August. Have a nice day.

Such replies can also specify how to handle urgent matters in the interim, other people to contact for specific problems, etc. Most vacation daemons keep track of whom they have sent canned replies to and refrain from sending the same person a second reply. The good ones also check to see if the incoming message was sent to a mailing list, and if so, do not send a canned reply at all. (People who send messages to large mailing lists during the summer probably do not want to get hundreds of replies detailing everyone's vacation plans.)

The author recently ran into a most extreme form of delivery processing when he sent an email message to a person who claims to get 600 messages a day. His identity will not be disclosed here, lest half the readers of this book also send him email. Let us call him John.

John has installed an email robot that checks every incoming message to see if it is from a new correspondent. If so, it sends back a canned reply explaining that John can no longer personally read all his email. Instead he has produced a personal FAQ (Frequently Asked Questions) document that answers many questions he is commonly asked. Normally, newsgroups have FAQs, not people.

John's FAQ gives his address, fax, and telephone numbers and tells how to contact his company. It explains how to get him as a speaker and describes where to get his papers and other documents. It also provides pointers to software he has written, a conference he is running, a standard he is the editor of, and so on. Perhaps this approach is necessary, but maybe a personal FAQ is the ultimate status symbol.

7.4.5. Email Privacy

When an email message is sent between two distant sites, it will generally transit dozens of machines on the way. Any of these can read and record the message for future use. Privacy is nonexistent, despite what many people think (Weisband and Reinig, 1995). Nevertheless, many people would like to be able to send email that can be read by the intended recipient and no one else: not their boss, not hackers, not even the government. This desire has stimulated several people and groups to apply the cryptographic principles we studied earlier to email to produce secure email. In the following sections we will study two widely used secure email systems, PGP and PEM. For additional information, see (Kaufman et al., 1995; Schneier, 1995; Stallings, 1995b; and Stallings, 1995c).

multiple newsgroups will contain all of their names. Because multiple names are allowed here, the *Followup-To:* header is needed to tell people where to post comments and reactions to put all of the subsequent discussion in one newsgroup.

The *Distribution:* header tells how far to spread the posting. It may contain one or more state or country codes, the name of a specific site or network, or "world."

The *Nntp-Posting-Host:* header is analogous to the RFC 822 *Sender:* header. It tells which machine actually posted the article, even if it was composed on a different machine (NNTP is the news protocol, described below).

The *References:* header indicates that this article is a response to an earlier article and gives the ID of that article. It is required on all follow-up articles and prohibited when starting a new discussion.

The *Organization:* header can be used to tell what company, university, or agency the poster is affiliated with. Articles that fill in this header often have a disclaimer at the end saying that if the article is goofy, it is not the organization's fault.

The *Lines:* header gives the length of the body. The header lines and the blank line separating the header from the body do not count.

The *Subject:* lines tie discussion threads together. Many news readers have a command to allow the user to see the next article on the current subject, rather than the next article that came in. Also, killfiles and kill commands use this header to know what to reject.

Finally, the *Summary:* is normally used to summarize the follow-up article. On follow-up articles, the *Subject:* header contains "Re: " followed by the original subject.

NNTP—Network News Transfer Protocol

Now let us look at how articles diffuse around the network. The initial algorithm just flooded articles onto every line within USENET. While this worked for a while, eventually the volume of traffic made this scheme impractical, so something better had to be worked out.

Its replacement was a protocol called **NNTP (Network News Transfer Protocol)**, which is defined in RFC 977. NNTP has something of the same flavor as SMTP, with a client issuing commands in ASCII and a server issuing responses as decimal numbers coded in ASCII. Most USENET machines now use NNTP.

NNTP was designed for two purposes. The first goal was to allow news articles to propagate from one machine to another over a reliable connection (e.g., TCP). The second goal was to allow users whose desktop computers cannot receive news to read news remotely. Both are widely used, but we will concentrate on how news articles spread out over the network using NNTP.

As mentioned above, two general approaches are possible. In the first one, news pull, the client calls one of its newsfeeds and asks for new news. In the

second one, news push, the newsfeed calls the client and announces that it has news. The NNTP commands support both of these approaches, as well as having people read news remotely.

To acquire recent articles, a client must first establish a TCP connection with port 119 on one of its newsfeeds. Behind this port is the NNTP daemon, which is either there all the time waiting for clients or is created on the fly as needed. After the connection has been established, the client and server communicate using a sequence of commands and responses. These commands and responses are used to ensure that the client gets all the articles it needs, but no duplicates, no matter how many newsfeeds it uses. The main ones used for moving articles between news daemons are listed in Fig. 7-56.

Command	Meaning
LIST	Give me a list of all newsgroups and articles you have
NEWGROUPS date time	Give me a list of newsgroups created after date/time
GROUP grp	Give me a list of all articles in grp
NEWNEWS grps date time	Give me a list of new articles in specified groups
ARTICLE id	Give me a specific article
POST	I have an article for you that was posted here
IHAVE id	I have article id. Do you want it?
QUIT	Terminate the session

Fig. 7-56. The principal NNTP commands for news diffusion.

The *LIST* and *NEWGROUPS* commands allow the client to find out which groups the server has. The former gives the complete list. The latter gives only those groups created after the date and time specified. If the client knows the list is long, it is more efficient for the client to keep track of what each of its newsfeeds has and just ask for updates. The responses to each of these commands is a list, in ASCII, one newsgroup per line, giving the name of the newsgroup, the number of the last article the server has, the number of the first article the server has, and a flag telling whether posting to this newsgroup is allowed.

Once the client knows which newsgroups the server has, it can begin asking about what articles the server has (e.g., for old newsgroups when *NEWGROUPS* is used). The *GROUP* and *NEWNEWS* commands are used for this purpose. Again, the former gives the full list and the latter gives only updates subsequent to the indicated date and time, normally the time of the last connection to this newsfeed. The first parameter may contain asterisks, meaning all of them. For example, *comp.os.** means all the newsgroups that start with the string *comp.os*.

After the client has assembled a complete list of which articles exist in which groups (or even before it has the full list), it can begin to ask for the articles it

needs using the *ARTICLE* command. Once all the required articles are in, the client can offer articles it has acquired from other newsfeeds using the *IHAVE* command and articles that were posted locally using the *POST* command. The server can accept or decline these, as it wishes. When the client is done, it can terminate the session using *QUIT*. In this way, each machine has complete control over which articles it gets from which newsfeeds, eliminating all duplicate articles.

As an example of how NNTP works, consider an information provider, *wholesome.net* that wants to avoid controversy at all costs, so the only newsgroups it offers are *soc.couples* and *misc.kids*. Nevertheless, management is open minded and willing to carry other newsgroups, provided they contain no material potentially offensive to anyone. Therefore, it wants to be informed of all newly created groups so it can make an informed decision for its customers. A possible scenario between *wholesome.com* acting as the client and its newsfeed, *feeder.com*, acting as the server, is shown in Fig. 7-57. This scenario uses the news pull approach (the client initiates the connection to ask for news). The remarks in parentheses are comments and not part of the NNTP protocol.

In this session, *wholesome.com* first asks if there is any news for *soc.couples*. When it is told there are two articles, it fetches both of them and stores them in *news/soc/couples* as separate files. Each file is named by its article number. Then *wholesome.com* asks about *misc.kids* and is told there is one article. It fetches that one and puts it in *news/misc/kids*.

Having gotten all the news about the groups it carries, it now checks for new groups and is told that two new groups have appeared since the last session. One of them looks promising, so its articles are fetched. The other looks scary, so it is not taken. (*Wholesome.com* has made a big investment in AI software to be able to figure out what to carry just by looking at the names.)

After having acquired all the articles it wants, *wholesome.com* offers *feeder.com* a new article posted by someone at its site. The offer is accepted and the article is transferred. Now *wholesome.com* offers another article, one that came from its other newsfeed. Since *feeder.com* already has this one, it declines. Finally, *wholesome.com* ends the session and releases the TCP connection.

The news push approach is similar. It begins with the newsfeed calling the machine that is to receive the news. The newsfeed normally keeps track of which newsgroups its customers subscribe to and begins by announcing its first article in the first of these newsgroups using the *IHAVE* command. The potential recipient then checks its tables to see whether it already has the article, and can accept or reject it. If the article is accepted, it is transmitted, followed by a line containing a period. Then the newsfeed advertises the second article, and so forth, until all the news has been transferred.

A problem with both news pull and news push is that they use stop and wait. Typically 100 msec are lost waiting for an answer to a question. With 100,000 or more news articles per day, this lost time adds up to a substantial overhead.

```
    S: 200 feeder.com NNTP server at your service(response to new connection)
C: NEWNEWS soc.couples 960901 030000 (any new news in soc.couples?)
    S: 230 List of 2 articles follows
    S: <13281@psyc.berkeley.edu>    (article 1 of 2 in soc.couples is from Berkeley)
    S: <162721@aol.com>             (article 2 of 2 in soc.couples is from AOL)
    S: .                            (end of list)
C: ARTICLE <13281@psyc.berkeley.edu>  (please give me the Berkeley article)
    S: 220 <13281@psyc.berkeley.edu> follows
    S: (entire article <13281@psyc.berkeley.edu> is sent here)
    S: .                            (end of article)
C: ARTICLE <162721@aol.com>    (please give me the AOL article)
    S: 220  <162721@aol.com> follows
    S: (entire article <162721@aol.com> is sent here)
    S: .                            (end of article)
C: NEWNEWS misc.kids 960901 030000    (any new news in misc.kids?)
    S: 230 List of 1 article follows
    S: <43222@bio.rice.edu>         (1 article from Rice)
    S: .                            (end of list)
C: ARTICLE <43222@bio.rice.edu>(please give me the Rice article)
    S: 220  <43222@bio.rice.edu> follows
    S: (entire article <43222@bio.rice.edu> is sent here)
    S: .                            (end of article)
C: NEWGROUPS 960901 030000
    S: 231 2 new groups follow
    S: rec.pets
    S: rec.nude
    S: .
C: NEWNEWS rec.pets 0 0             (list everything you have)
    S: 230 List of 1 article follows
    S: <124@fido.net>               (1 article from fido.net)
    S: .                            (end of list)
C: ARTICLE <124@fido.net>           (please give me the fido.net article)
    S: 220 <124@fido.net> follows
    S: (entire article is sent here)
    S: .
C: POST
    S: 340                          (please send your posting)
C:  (article posted on wholesome.com sent here)
    S: 240                          (article received)
C: IHAVE <5321@foo.com>
    S: 435                          (I already have it, please do not send it)
C: QUIT
    S: 205                          (Have a nice day)
```

Fig. 7-57. How *wholesome.com* might acquire news articles from its newsfeed.

Readers are encouraged to try this scenario personally (preferably from a UNIX system, because some other systems do not return the connection status). Be sure to note the spaces and the protocol version on the *GET* line, and the blank line following the *GET* line. As an aside, the actual text that will be received will differ from what is shown in Fig. 7-60 for three reasons. First, the example output here has been abridged and edited to make it fit on one page. Second, it has been cleaned up somewhat to avoid embarrassing the author, who no doubt expected thousands of people to examine the formatted page, but zero people to scrutinize the HTML that produced it. Third, the contents of the page are constantly being revised. Nevertheless, this example should give a reasonable idea of how HTTP works.

What the example shows is the following. The client, in this case a person, but normally a browser, first connects to a particular host and then sends a command asking for a particular page and specifying a particular protocol and version to use (HTTP/1.0). On line 7, the server responds with a status line telling the protocol it is using (the same as the client) and the code 200, meaning OK. This line is followed by an RFC 822 MIME message, of which five of the header lines are shown in the figure (several others have been omitted to save space). Then comes a blank line, followed by the message body. For sending a picture, the *Content-Type* field might be

```
Content-Type: Image/GIF
```

In this way, the MIME types allow arbitrary objects to be sent in a standard way. As an aside, the MIME *Content-Transfer-Encoding* header is not needed because TCP allows arbitrary byte streams, even pictures, to be sent without modification. The meaning of the commands within angle brackets used in the sample page will be discussed later in this chapter.

Not all servers speak HTTP. In particular, many older servers use the FTP, Gopher, or other protocols. Since a great deal of useful information is available on FTP and Gopher servers, one of the design goals of the Web was to make this information available to Web users. One solution is to have the browser use these protocols when speaking to an FTP or Gopher server. Some of them, in fact, use this solution, but making browsers understand every possible protocol makes them unnecessarily large.

Instead, a different solution is often used: proxy servers (Luotonen and Altis, 1994). A **proxy server** is a kind of gateway that speaks HTTP to the browser but FTP, Gopher, or some other protocol to the server. It accepts HTTP requests and translates them into, say, FTP requests, so the browser does not have to understand any protocol except HTTP. The proxy server can be a program running on the same machine as the browser, but it can also be on a free-standing machine somewhere in the network serving many browsers. Figure 7-61 shows the difference between a browser that can speak FTP and one that uses a proxy.

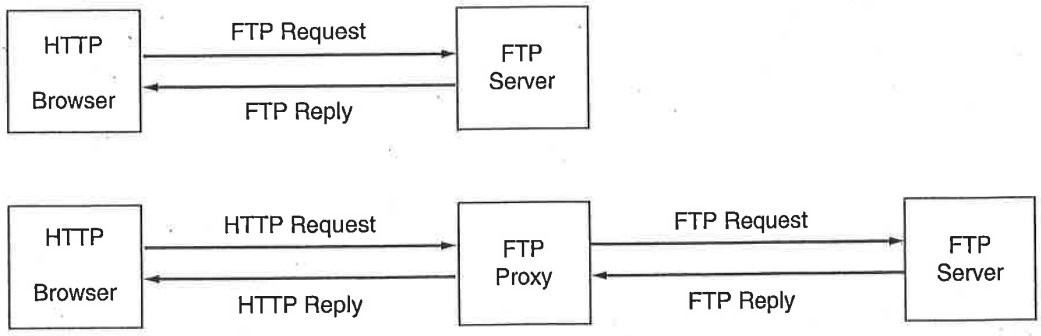


Fig. 7-61. (a) A browser that speaks FTP. (b) A browser that does not.

Often users can configure their browsers with proxies for protocols that the browsers do not speak. In this way, the range of information sources to which the browser has access is increased.

In addition to acting as a go-between for unknown protocols, proxy servers have a number of other important functions, such as caching. A caching proxy server collects and keeps all the pages that pass through it. When a user asks for a page, the proxy server checks to see if it has the page. If so, it can check to see if the page is still current. In the event that the page is still current, it is passed to the user. Otherwise, a new copy is fetched.

Finally, an organization can put a proxy server inside its firewall to allow users to access the Web, but without giving them full Internet access. In this configuration, users can talk to the proxy server, but it is the proxy server that contacts remote sites and fetches pages on behalf of its clients. This mechanism can be used, for example, by high schools, to block access to Web sites the principal feels are inappropriate for tender young minds.

For information about one of the more popular Web servers (NCSA's HTTP daemon) and its performance, see (Katz et al., 1994; and Kwan et al., 1995).

HTTP—HyperText Transfer Protocol

The standard Web transfer protocol is **HTTP (HyperText Transfer Protocol)**. Each interaction consists of one ASCII request, followed by one RFC 822 MIME-like response. Although the use of TCP for the transport connection is very common, it is not formally required by the standard. If ATM networks become reliable enough, the HTTP requests and replies could be carried in AAL 5 messages just as well.

HTTP is constantly evolving. Several versions are in use and others are under development. The material presented below is relatively basic and is unlikely to change in concept, but some details may be a little different in future versions.

The HTTP protocol consists of two fairly distinct items: the set of requests from browsers to servers and the set of responses going back the other way. We will now treat each of these in turn.

All the newer versions of HTTP support two kinds of requests: simple requests and full requests. A simple request is just a single *GET* line naming the page desired, without the protocol version. The response is just the raw page, with no headers, no MIME, and no encoding. To see how this works, try making a Telnet connection to port 80 of *www.w3.org* (as shown in the first line of Fig. 7-60) and then type

```
GET /hypertext/WWW/TheProject.html
```

but without the HTTP/1.0 this time. The page will be returned with no indication of its content type. This mechanism is needed for backward compatibility. Its use will decline as browsers and servers based on full requests become standard.

Full requests are indicated by the presence of the protocol version on the *GET* request line, as in Fig. 7-60. Requests may consist of multiple lines, followed by a blank line to indicate the end of the request, which is why the blank line was needed in Fig. 7-60. The first line of a full request contains the command (of which *GET* is but one of the possibilities), the page desired, and the protocol/version. Subsequent lines contain RFC 822 headers.

Although HTTP was designed for use in the Web, it has been intentionally made more general than necessary with an eye to future object-oriented applications. For this reason, the first word on the full request line is simply the name of the **method** (command) to be executed on the Web page (or general object). The built-in methods are listed in Fig. 7-62. When accessing general objects, additional object-specific methods may also be available. The names are case sensitive, so, *GET* is a legal method but *get* is not.

Method	Description
GET	Request to read a Web page
HEAD	Request to read a Web page's header
PUT	Request to store a Web page
POST	Append to a named resource (e.g., a Web page)
DELETE	Remove the Web page
LINK	Connects two existing resources
UNLINK	Breaks an existing connection between two resources

Fig. 7-62. The built-in HTTP request methods.

The *GET* method requests the server to send the page (by which we mean object, in the most general case), suitably encoded in MIME. However, if the

GET request is followed by an *If-Modified-Since* header, the server only sends the data if it has been modified since the date supplied. Using this mechanism, a browser that is asked to display a cached page can conditionally ask for it from the server, giving the modification time associated with the page. If the cache page is still valid, the server just sends back a status line announcing that fact, thus eliminating the overhead of transferring the page again.

The *HEAD* method just asks for the message header, without the actual page. This method can be used to get a page's time of last modification, to collect information for indexing purposes, or just to test a URL for validity. Conditional *HEAD* requests do not exist.

The *PUT* method is the reverse of *GET*: instead of reading the page, it writes the page. This method makes it possible to build a collection of Web pages on a remote server. The body of the request contains the page. It may be encoded using MIME, in which case the lines following the *PUT* might include *Content-Type* and authentication headers, to prove that the caller indeed has permission to perform the requested operation.

Somewhat similar to *PUT* is the *POST* method. It too bears a URL, but instead of replacing the existing data, the new data is "appended" to it in some generalized sense. Posting a message to a news group or adding a file to a bulletin board system are examples of appending in this context. It is clearly the intention here to have the Web take over the functionality of the USENET news system.

DELETE does what you might expect: it removes the page. As with *PUT*, authentication and permission play a major role here. There is no guarantee that *DELETE* succeeds, since even if the remote HTTP server is willing to delete the page, the underlying file may have a mode that forbids the HTTP server from modifying or removing it.

The *LINK* and *UNLINK* methods allow connections to be established between existing pages or other resources.

Every request gets a response consisting of a status line, and possibly additional information (e.g., all or part of a Web page). The status line can bear the code 200 (OK), or any one of a variety of error codes, for example 304 (not modified), 400 (bad request), or 403 (forbidden).

The HTTP standards describe message headers and bodies in considerable detail. Suffice it to say that these are very close to RFC 822 MIME messages, so we will not look at them here.

7.6.3. Writing a Web Page in HTML

Web pages are written in a language called **HTML (HyperText Markup Language)**. HTML allows users to produce Web pages that include text, graphics, and pointers to other Web pages. We will begin our study of HTML with these pointers, since they are the glue that holds the Web together.

URLs—Uniform Resource Locators

We have repeatedly said that Web pages may contain pointers to other Web pages. Now it is time to see how these pointers are implemented. When the Web was first created, it was immediately apparent that having one page point to another Web page required mechanisms for naming and locating pages. In particular, there were three questions that had to be answered before a selected page could be displayed:

1. What is the page called?
2. Where is the page located?
3. How can the page be accessed?

If every page were somehow assigned a unique name, there would not be any ambiguity in identifying pages. Nevertheless, the problem would not be solved. Consider a parallel between people and pages. In the United States, almost everyone has a social security number, which is a unique identifier, as no two people have the same one. Nevertheless, armed only with a social security number, there is no way to find the owner's address, and certainly no way to tell whether you should write to the person in English, Spanish, or Chinese. The Web has basically the same problems.

The solution chosen identifies pages in a way that solves all three problems at once. Each page is assigned a **URL (Uniform Resource Locator)** that effectively serves as the page's worldwide name. URLs have three parts: the protocol (also called a scheme), the DNS name of the machine on which the page is located, and a local name uniquely indicating the specific page (usually just a file name on the machine where it resides). For example, the URL for the author's department is

```
http://www.cs.vu.nl/welcome.html
```

This URL consists of three parts: the protocol (*http*), the DNS name of the host (*www.cs.vu.nl*), and the file name (*welcome.html*), with certain punctuation separating the pieces.

Many sites have certain shortcuts for file names built in. For example, *~user/* might be mapped onto *user*'s WWW directory, with the convention that a reference to the directory itself implies a certain file, say, *index.html*. Thus the author's home page can be reached at

```
http://www.cs.vu.nl/~ast/
```

even though the actual file name is different. At many sites, a null file name default's to the organization's home page.

Now it should be clear how hypertext works. To make a piece of text clickable, the page writer must provide two items of information: the clickable text to

be displayed and the URL of the page to go to if the text is selected. When the text is selected, the browser looks up the host name using DNS. Now armed with the host's IP address, the browser then establishes a TCP connection to the host. Over that connection, it sends the file name using the specified protocol. Bingo. Back comes the page. This is precisely what we saw in Fig. 7-60.

This URL scheme is open-ended in the sense that it is straightforward to have protocols other than HTTP. In fact, URLs for various other common protocols have been defined, and many browsers understand them. Slightly simplified forms of the more common ones are listed in Fig. 7-63.

Name	Used for	Example
http	Hypertext (HTML)	http://www.cs.vu.nl/~ast/
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	/usr/suzanne/prog.c
news	News group	news:comp.os.minix
news	News article	news:AA0134223112@cs.utah.edu
gopher	Gopher	gopher://gopher.tc.umn.edu/11/Libraries
mailto	Sending email	mailto:kim@acm.org
telnet	Remote login	telnet://www.w3.org:80

Fig. 7-63. Some common URLs.

Let us briefly go over the list. The *http* protocol is the Web's native language, the one spoken by HTTP servers. It supports all the methods of Fig. 7-62, as well as whatever object-specific methods are needed.

The *ftp* protocol is used to access files by FTP, the Internet's file transfer protocol. FTP has been around more than two decades and is well entrenched. Numerous FTP servers all over the world allow people anywhere on the Internet to log in and download whatever files have been placed on the FTP server. The Web does not change this; it just makes obtaining files by FTP easier, as FTP has a somewhat arcane interface. In due course, FTP will probably vanish, as there is no particular advantage for a site to run an FTP server instead of an HTTP server, which can do everything that the FTP server can do, and more (although there are some arguments about efficiency).

It is possible to access a local file as a Web page, either by using the *file* protocol, or more simply, by just naming it. This approach is similar to using FTP but does not require having a server. Of course, it only works for local files.

The *news* protocol allows a Web user to call up a news article as though it were a Web page. This means that a Web browser is simultaneously a news reader. In fact, many browsers have buttons or menu items to make reading USENET news even easier than using standard news readers.

Two formats are supported for the *news* protocol. The first format specifies a newsgroup and can be used to get a list of articles from a preconfigured news site. The second one requires the identifier of a specific news article to be given, in this case *AA0134223112@cs.utah.edu*. The browser then fetches the given article from its preconfigured news site using the NNTP protocol.

The *gopher* protocol is used by the Gopher system, which was designed at the University of Minnesota and named after the school's athletic teams, the Golden Gophers (as well as being a slang expression meaning "go for", i.e., go fetch). Gopher predates the Web by several years. It is an information retrieval scheme, conceptually similar to the Web itself, but supporting only text and no images. When a user logs into a Gopher server, he is presented with a menu of files and directories, any of which can be linked to another Gopher menu anywhere in the world.

Gopher's big advantage over the Web is that it works very well with 25×80 ASCII terminals, of which there are still quite a few around, and because it is text based, it is very fast. Consequently, there are thousands of Gopher servers all over the world. Using the *gopher* protocol, Web users can access Gopher and have each Gopher menu presented as a clickable Web page. If you are not familiar with Gopher, try the example given in Fig. 7-63 or have your favorite Web search engine look for "gopher."

Although the example given does not illustrate it, it is also possible to send a complete query to a Gopher server using the *gopher+* protocol. What is displayed is the result of querying the remote Gopher server.

The last two protocols do not really have the flavor of fetching Web pages, and are not supported by all browsers, but are useful anyway. The *mailto* protocol allows users to send email from a Web browser. The way to do this is to click on the OPEN button and specify a URL consisting of *mailto:* followed by the recipient's email address. Most browsers will respond by popping up a form containing slots for the subject and other header lines and space for typing the message.

The *telnet* protocol is used to establish an on-line connection to a remote machine. It is used the same way as the Telnet program, which is not surprising, since most browsers just call the Telnet program as a helper application. As an exercise, try the scenario of Fig. 7-60 again, but now using a Web browser.

In short, the URLs have been designed to not only allow users to navigate the Web, but to deal with FTP, news, Gopher, email, and telnet as well, making all the specialized user interface programs for those other services unnecessary, and thus integrating nearly all Internet access into a single program, the Web browser. If it were not for the fact that this scheme was designed by a physics researcher, it could easily pass for the output of some software company's advertising department.

Despite all these nice properties, the growing use of the Web has turned up an inherent weakness in the URL scheme. A URL points to one specific host. For

pages that are heavily referenced, it is desirable to have multiple copies far apart, to reduce the network traffic. The trouble is that URLs do not provide any way to reference a page without simultaneously telling where it is. There is no way to say: "I want page xyz, but I do not care where you get it." To solve this problem and make it possible to replicate pages, the IETF is working on a system of **URIs** (**Universal Resource Identifiers**). A URI can be thought of as a generalized URL. This topic is the subject of much current research.

Although we have discussed only absolute URLs here, relative URLs also exist. The difference is analogous to the difference between the absolute file name */usr/ast/foobar* and just *foobar* when the context is unambiguously defined.

HTML—HyperText Markup Language

Now that we have a good idea of how URLs work, it is time to look at HTML itself. HTML is an application of ISO standard 8879, **SGML** (**Standard Generalized Markup Language**), but specialized to hypertext and adapted to the Web.

As mentioned earlier, HTML is a markup language, a language for describing how documents are to be formatted. The term "markup" comes from the old days when copyeditors actually marked up documents to tell the printer—in those days, a human being—which fonts to use, and so on. Markup languages thus contain explicit commands for formatting. For example, in HTML, <B> **means start boldface mode, and** </B> means leave boldface mode. The advantage of a markup language over one with no explicit markup is that writing a browser for it is straightforward: the browser simply has to understand the markup commands. TeX and troff are other well-known examples of markup languages.

Documents written in a markup language can be contrasted to documents produced with a WYSIWYG (What You See Is What You Get) word processor, such as MS-Word® or WordPerfect®. These systems may store their files with hidden embedded markup so they can reproduce them later, but not all of them work this way. Word processors for the Macintosh, for example, keep the formatting information in separate data structures, not as commands embedded in the user files.

By embedding the markup commands within each HTML file and standardizing them, it becomes possible for any Web browser to read and reformat any Web page. Being able to reformat Web pages after receiving them is crucial because a page may have been produced full screen on a 1024×768 display with 24-bit color but may have to be displayed in a small window on a 640×480 screen with 8-bit color. Proprietary WYSIWYG word processors cannot be used on the Web because their internal markup languages (if any) are not standardized across vendors, machines and operating systems. Also, they do not handle reformatting for different-sized windows and different resolution displays. However, word processing program can offer the option of saving documents in HTML instead of in the vendor's proprietary format, and some of them already do.